UBM

Learn today. Design tomorrow.

ESD

EMBEDDED SYSTEMS DESIGN

The Official Publication of The Embedded Systems Conferences and Embedded.com

VOLUME 23, NUMBER 4

MAY 2010

Using an FPGA to test a PLL band calibration algorithm, 23

Object-code generation fix, 29

Ganssle: More on test driven development, 33

Protect against malicious software

16

Learn today. Design tomorrow.

ESC

Embedded Systems Conference Chicago

Donald E. Stevens Convention Center, Rosemont, IL

Conference: June 7–9, 2010 • Expo: June 8–9, 2010

One Size Doesn't Fit All

- File Systems
- USB Stacks
- Bootloaders
- Windows Drivers
- Embedded Development Services

With HCC, you can choose a file system that's right for your application. HCC products run with the broadest range of CPUs and memory devices, with or without an operating system.

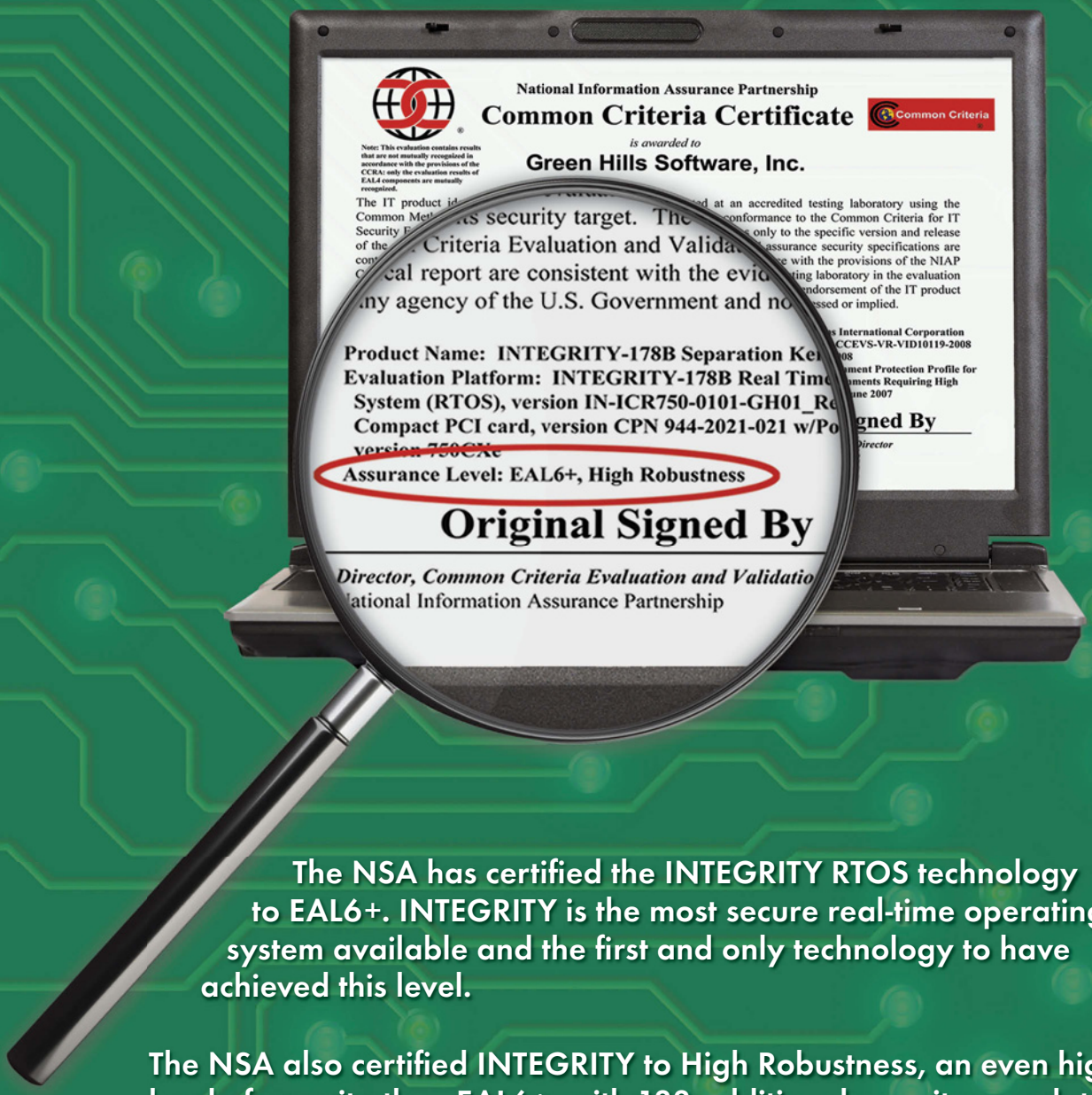
THE MOST COMPREHENSIVE PORTFOLIO OF FILE SYSTEMS FOR EMBEDDED APPLICATIONS

HCC-Embedded
FILE SYSTEMS WITH A DIFFERENCE

www.hcc-embedded.com • info@hcc-embedded.com

HCC embedded

INTEGRITY RTOS has it. No one else does.


National Information Assurance Partnership
Common Criteria Certificate
Common Criteria
is awarded to
Green Hills Software, Inc.
Product Name: INTEGRITY-178B Separation Ke
Evaluation Platform: INTEGRITY-178B Real Time
System (RTOS), version IN-ICR750-0101-GH01_Re
Compact PCI card, version CPN 944-2021-021 w/Po
Assurance Level: EAL6+, High Robustness
Original Signed By
Director, Common Criteria Evaluation and Validatio
National Information Assurance Partnership

The NSA has certified the INTEGRITY RTOS technology to EAL6+. INTEGRITY is the most secure real-time operating system available and the first and only technology to have achieved this level.

The NSA also certified INTEGRITY to High Robustness, an even higher level of security than EAL6+, with 133 additional security mandates over and above the 161 required for EAL6+.

When security is required, Green Hills Software's INTEGRITY RTOS technology is the only option.

Copyright © 2010 Green Hills Software, Inc. Green Hills, the Green Hills logo, and INTEGRITY are trademarks of Green Hills Software, Inc. in the U.S. and/or internationally. All other trademarks are the property of their respective owners.

Green Hills® SOFTWARE, INC.

www.ghs.com

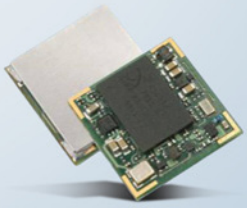
The Newest Products For Your Newest Designs

Embed Your Innovation into the Market Place.

XBee® and XBee-PRO® ZB Adapters
mouser.com/digixbeeadapter

SocketModem® iCell / Cell Embedded Wireless Modems
mouser.com/multitechicell/

ZICM2410P2 MeshConnect™ Module
mouser.com/celzicm2410p2



A1084-B GPS receiver module
mouser.com/vincotecha1084b

WARNING: Designing with Hot, New Products May Cause a Time-to-Market Advantage.

The newest embedded and wireless products and technologies make designing even more fun. Experience Mouser's time-to-market advantage with no minimums and same-day shipping of the newest products from more than 400 leading suppliers.

MOUSER ELECTRONICS®
a tti company

mouser.com **(800) 346-6873**

Mouser and Mouser Electronics are registered trademarks of Mouser Electronics, Inc. Other products, logos, and company names mentioned herein, may be trademarks of their respective owners.

Learn today. Design tomorrow.

ESD

EMBEDDED SYSTEMS DESIGN

VOLUME 23, NUMBER 4
MAY 2010

16

Cover Feature:
Bullet-proofing your software design

BY NAT HILLARY

Applying secure programming standards and methodology can reduce vulnerabilities in software.

23 **How to use an FPGA to test a PLL band calibration algorithm**

BY RUSSELL MOHN

Prototyping an ASIC design first on an FPGA is not only useful for verification but allows more room for algorithm experimentation.

29 **Dealing with misbehaving tools**

BY ANDERS HOLMBERG

When errors in object code are generated by your tools, such as the compiler, assembler, and linker, try this novel approach to the "update tool or update source" dilemma.

EMBEDDED SYSTEMS DESIGN (ISSN 1558-2493) print; (ISSN 1558-2507 PDF-electronic) is published 10 times a year as follows: Jan/Feb, March, April, May, June, July/August, Sept., Oct., Nov., Dec. by the EE Times Group, 600 Harrison Street, 5th floor, San Francisco, CA 94107, (415) 947-6000. Please direct advertising and editorial inquiries to this address. SUBSCRIPTION RATE for the United States is $55 for 10 issues. Canadian/Mexican orders must be accompanied by payment in U.S. funds with additional postage of $6 per year. All other foreign subscriptions must be prepaid in U.S. funds with additional postage of $15 per year for surface mail and $40 per year for airmail. POSTMASTER: Send all changes to EMBEDDED SYSTEMS DESIGN, P.O. Box 3404, Northbrook, IL 60065-9468. For customer service, telephone toll-free (877) 676-9745. Please allow four to six weeks for change of address to take effect. Periodicals postage paid at San Francisco, CA and additional mailing offices. EMBEDDED SYSTEMS DESIGN is a registered trademark owned by the parent company, EE Times Group. *All material published in EMBEDDED SYSTEMS DESIGN is copyright © 2010 by EE Times Group. All rights reserved. Reproduction of material appearing in EMBEDDED SYSTEMS DESIGN is forbidden without permission.*

COLUMNS

programming pointers 9

Alternative models for memory-mapped devices

BY DAN SAKS

Traditional techniques for communicating with hardware devices can be inconvenient and error-prone. Here are ways to make them simpler and robust.

break points 33

An interview with James Grenning, Part 2

BY JACK G. GANSSLE

Is test driven development viable for embedded systems? Ganssle continues to grill Grenning on TDD.

DEPARTMENTS

#include 5

Virtualization extends down to mobile devices

BY RICHARD NASS

The BOM on a virtualized handset can be reduced significantly over a more traditional design.

parity bit 7

marketplace 32

IN PERSON

ESC Chicago
June 7–9, 2010
esc-chicago.techinsightsevents.com/

ESC India
July 21–23, 2010
www.esc-india.com/

ESC Boston
September 20–23, 2010
www.embedded.com/esc/boston

Embedded Live
October 20–21, 2010
www.embedded.co.uk

ESC Silicon Valley
May 2–5, 2011
www.embedded.com/esc/sv

ONLINE

www.embedded.com

BUILD it [Reliably]

With Express Logic's award-winning BenchX® IDE or use tools from over 20 commercial offerings including those from ARM, Freescale, Green Hills, IAR, Microchip, MIPS, Renesas, and Wind River.

RUN it [Fast]

With Express Logic's small, fast, royalty-free and industry leading ThreadX® RTOS, NetX™ TCP/IP stack, FileX® FAT file system, and USBX™ USB stack.

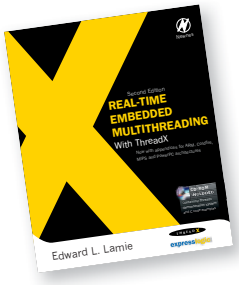
ANALYZE it [Easily]

With Express Logic's graphical TraceX® event analysis tool, and new StackX™ stack usage analysis tool. See exactly what is happening in your system, which is essential for both debugging and optimization.

SHIP it [Confidently]

No matter what "it" is you're developing, Express Logic's solutions will help you build it, analyze it, run it, and ship it better and in less time. Join the success of over 600,000,000 deployed products using Express Logic's ThreadX!

BENCHX THREADX TRACEX STACKX



expresslogic

For a free evaluation copy, visit www.rtos.com • 1-888-THREADX

ThreadX, BenchX, TraceX and FileX are a registered trademarks of Express Logic, Inc. All other trademarks are the property of their respective owners.

EMBEDDED SYSTEMS DESIGN

Editorial Director
Richard Nass
(201) 288-1904
rich.nass@ubm.com

Managing Editor
Susan Rambo
susan.rambo@ubm.com

Contributing Editors
Michael Barr, John Canosa,
Jack W. Crenshaw, Jack G. Ganssle,
Dan Saks, Larry Mittag

Art Director
Debee Rommel
debee.rommel@ubm.com

European Correspondent
Colin Holland
colin.holland@ubm.com

Embedded.com Site Editor
Bernard Cole
bccole@acm.org

Production Director
Donna Ambrosino
dambrosino@ubm-us.com

Subscription Customer Service
P.O. Box 2165, Skokie, IL 60076
(800) 577-5356 (toll free)
Fax: (847) 763-9606
embeddedsystemsdesign@halldata.com
www.customerserviceesp.com

Article Reprints, E-prints, and Permissions
Mike O'Brien
Wright's Reprints
(877) 652-5295 (toll free)
(281) 419-5725 ext.117
Fax: (281) 419-5712
www.wrightsreprints.com/reprints/index.cfm?magid=2210

Publisher
David Blaza
(415) 947-6929
david.blaza@ubm.com

Editorial Review Board
Michael Barr, Jack W. Crenshaw,
Jack G. Ganssle, Bill Gatliff,
Nigel Jones, Niall Murphy, Dan Saks,
Miro Samek

EE Times Group

Corporate—EE Times Group

Paul Miller	*Chief Executive Officer*
Felicia Hamerman	*Group Marketing Director*
Brent Pearson	*Chief Information Officer*
Jean-Marie Enjuto	*Financial Director*
Amandeep Sandhu	*Manager Audience Engagement*
Barbara Couchois	*Vice President Sales Ops*

UBM

Corporate—UBM LLC

Marie Myers	*Senior Vice President, Manufacturing*
Pat Nohilly	*Senior Vice President, Strategic Development and Business Administration*

#include

BY Richard Nass

Virtualization extends down to mobile devices

It wasn't too long ago that you had to explain to designers what the term "virtualization" was all about. That's generally not the case anymore. In fact, the number of systems that are "virtualized" as part of the design process has increased considerably.

In most cases, the systems that were candidates for virtualization were those of the high-performance variety. Today, you can make an argument to go the virtual route for almost any type of embedded system, including mobile devices. The technology that's been deployed for years in enterprise data centers can be driven down into consumer devices.

While traditional virtualization often concentrates on a high-end (even multicore) processor and chipset, mobile virtualization tends to look at the paradigm a little differently. For example, it's no surprise to see multiple chips consolidated into a unified processor, with dedicated DRAM, glue logic, and so on. At the same time, multiple functions can be ported to that single virtualized processor. While potentially increasing performance, there's a big upside in the reduced number of interconnects required. The single virtualized processor also reduces the overall memory footprint required by the system.

A side-by-side comparison of a handset that was virtualized by Open Kernel Labs (OK Labs, for short) shows a significant cost savings, up to 46% over a similar, nonvirtualized design. The biggest savings comes from being able to eliminate the applications processor, which could cost as much as $30, depending on the required performance, functionality, and feature set of the handset.

Eliminating that apps processor reduces the memory footprint, thereby savings a few dollars. The only additional cost to the BOM comes in the way of the hypervisor needed to do the virtualization, which is where OK Labs comes in. They provide that hypervisor software.

One example of a handset that was virtualized using the OK Labs technology is the Motorola Evoke. It employs an embedded hypervisor from OK Labs, called the OKL4 (note that the Motorola Evoke will be the object of an upcoming Tear Down article, where we really get into the nuts and bolts of how the handset was designed, with a particular focus on the virtualization aspect).

The bottom line is that you shouldn't dismiss virtualization technology as just being for high-end devices. You could be missing an opportunity to reduce your BOM and ultimately simplify your design, especially if your product lends itself to multiple versions/family members.

Richard Nass (rich.nass@ubm.com) is the editorial director of* Embedded Systems Design *magazine, Embedded.com, and the Embedded Systems Conference.

Richard Nass
rich.nass@ubm.com

336 Volts of Green Engineering

MEASURE IT – FIX IT

Developing a commercially viable fuel cell vehicle has been a significant challenge because of the considerable expense of designing and testing each new concept. With NI LabVIEW graphical programming and NI CompactRIO hardware, Ford quickly prototyped fuel cell control unit iterations, resulting in the world's first fuel cell plug-in hybrid.

MEASURE IT

Acquire
Acquire and measure data from any sensor or signal

Analyze
Analyze and extract information with signal processing

Present
Present data with HMIs, Web interfaces, and reports

FIX IT

Design
Design optimized control algorithms and systems

Prototype
Prototype designs on ready-to-run hardware

Deploy
Deploy to the hardware platform you choose

Ford is just one of many customers using the NI graphical system design platform to improve the world around them. Engineers and scientists in virtually every industry are creating new ways to measure and fix industrial machines and processes so they can do their jobs better and more efficiently. And, along the way, they are creating innovative solutions to address some of today's most pressing environmental issues.

>> Download the Ford technical case study at **ni.com/336** **800 258 7018**

NATIONAL INSTRUMENTS™

©2009 National Instruments. All rights reserved. CompactRIO, LabVIEW, National Instruments, NI, and ni.com are trademarks of National Instruments.
Other product and company names listed are trademarks or trade names of their respective companies. 1121

Debating test driven development

With respect to the guest *(Jack Ganssle, "An interview with James Grenning," April 2010, p.35, www.embedded.com/224200702)*, he is an example of the kind of software person who focuses too much on the software itself and not enough on the whole system. Consider: "The code specifies the behavior of the executable program in all its necessary detail."

That's a true statement but a misleading one. Only the target processor can read the code to the level of precision needed to answer key questions, like, "What is the maximum travel of this actuator?" or, "What is the maximum RPM of this motor?" Therefore, human reading of source code is no substitute for a concise, accurate, top-level specification, which feeds both implementation and system test activities.

Here's an example. Some years ago, I heard about a software-related incident at an automated sawmill. The target processor was 16 bits, and the software was calibrated in .001-inch units. The first log that came through the system with diameter greater than 32.767 inches caused a reversal of some actuator that broke stuff and injured the operator. So code may equal design, but in this case something more explicit was needed. I daresay that TDD-style unit testing would not have caught that problem either.

One difference between embedded and other software is that physical systems often have complex requirements of their own. Those requirements need to be captured independently of the code, and need system-level testing independent of any in-code specification.

I think a certain personality type likes to pick apart the waterfall method, and there are obvious arguments against it. But so far, it is the best tool going for managing complexity and risk in real-world projects. You can use TDD or OOD or whatever you need to use inside your waterfall block, but your block has to be correct in relation to the other blocks of the project.

— Larry Martin
Owner, www.GlueLogix.com

One difference between embedded and other software is that physical systems often have complex requirements of their own.

TDD is a way to ensure that some analysis and design is being addressed "up-front", but this article didn't cover all my concerns.

I've seen a lot of projects run into trouble, when integration occurs, due to a lack of planning for integration of the units. TDD doesn't seem to address this.

I also believe it's very beneficial to have independence in testing, even at the unit test level. It is too common for developers to write tests to prove their code correct rather than to prove it incorrect. Under time pressure, this becomes a subconscious driver. It would be really easy to write "soft" tests in TDD as well. Addressing this would probably result in developers complaining about BTUF (Big Testing Up Front).

—lwriemen
Chief Frog

I think we all agree that the product must be tested as whole in its intended environment. Or...? The requirements may be flawed or incomplete, numerous research have labelled poor requirements as the most common cause for disaster.

Also, no matter how loose coupling your code has, there might be unforseen dependencies arising when the code is put together. All modules in your project will also share the same system resources. And there will always be the top-down dependancy from main thread -> code modules -> objects/functions. So the code, too, needs to be tested in its intended software environment, just like the final product needs to be tested in the intended "real world" environment.

None of the this excludes TDD. But of course TDD can't be used as a replacement for final verification/validation, that would just be plain stupid, for the above mentioned reasons. I really don't hope that's the case...

I'm going to be mean and shoot down [Larry Martin's] example [about the automated sawmill]. Any decent static analyzer (and good compilers as well) would have attacked that particular bug saying something like "implicit conversion between signed and unsigned int." If you scratch your head for

CONTINUED ON PAGE 32

United for a better world

NEC Electronics Corporation and Renesas Technology Corp. have combined their businesses to launch

Renesas Electronics Corporation.

Renesas Electronics has integrated the microcontroller, power and analog semiconductors, as well as the advanced technologies and strong experience of both companies, to provide system solutions for the global market. As the #1 supplier of microcontrollers in the world*, the new Renesas Electronics has a corporate vision to harness the power of semiconductors to realize intelligent devices that can enable a sustainable planet and improve the lives of both current and future generations. Together we can realize a society where life can be enjoyed in harmony with the planet.

* Source: Gartner "Semiconductor Applications Worldwide Annual Market Share: Database" 2 April, 2009. This is the 2008 ranking based on revenue.

www.renesas.com

Renesas Electronics

RENESAS

By Dan Saks

Alternative models for memory-mapped devices

Device drivers typically communicate with hardware devices through device registers. A driver sends commands or data to a device by storing into device registers, or retrieves status information or data from a device by reading from device registers.

Many processors use *memory-mapped I/O*, which maps device registers to fixed addresses in the conventional memory space. To a C or C++ programmer, a memory-mapped device register looks very much like an ordinary data object. Programs can use built-in operators such as assignment to move values to or from memory-mapped device registers.

Some processors use *port-mapped I/O*, which maps device registers to addresses in a separate address space, apart from the conventional memory space. Port-mapped I/O usually requires special machine instructions, such as the `in` and `out` instructions of the Intel x86 processors, to move data to or from device registers. To a C or C++ programmer, port-mapped device registers don't look like ordinary memory.

The C and C++ standards say nothing about port-mapped I/O. Programs that perform port-mapped I/O must use nonstandard, platform-specific language or library extensions, or worse, assembly code. On the other hand, programs can perform memory-mapped I/O using only standard language features. Fortunately, port-mapped I/O appears to be gradually fading away, and fewer programmers need to fuss with it.

Traditional techniques for communicating with hardware devices can be inconvenient and error-prone. Here are ways to make them simpler and robust.

Several years ago, I wrote a series of articles on accessing memory-mapped device registers using C and C++.[1, 2, 3] At the time, I focused on how to set up pointers (in C or C++) or references (in C++) to enable access to memory-mapped registers. I followed those articles with another discussing some alternatives for choosing the types that represent the registers themselves.[4]

In the years since then, I've had many discussions with readers and conference attendees who use other techniques for representing device registers. I find that many programmers are still using approaches that are inconvenient and error-prone. This month, I'll compare some popular alternatives, focusing more on the interface design issues than on the implementation details.

MAPPING MEMORY THE OLD-FASHIONED WAY

Let's consider a machine with a variety of devices, including a programmable timer and a couple of UARTs (serial ports), each of which employs a small collection of device registers. The timer registers start at location 0xFFFF6000. The registers for UART0 and UART1 start at 0xFFFFD000 and 0xFFFFE000, respectively.

For simplicity, let's assume that every device register is a four-byte word aligned to an address that's a multiple of four, so that you can manipulate each device register as an `unsigned int`. Many programmers prefer to use an exact-width type such as `uint32_t`. (Types such as `uint32_t` are defined in the C99 header `<stdint.h>`.)[5]

I prefer to use a symbolic type whose name conveys the meaning of the type rather than its physical extent, such as:

Dan Saks is president of Saks & Associates, a C/C++ training and consulting company. For more information about Dan Saks, visit his website at www.dansaks.com. Dan also welcomes your feedback: e-mail him at dan@dansaks.com.

Innovative Intelligent Integration

SmartFusion™

FPGA + ARM®Cortex™-M3 + Programmable Analog

© 2010 Actel Corporation. All rights reserved.

Actel POWER MATTERS

Actel POWER MATTERS

Get Smart, visit: **www.actel.com/smartfusion**

```
typedef uint32_t device_register;
```

Device registers are actually volatile entities—they may change state in ways that compilers can't detect.[6,7] I often include the volatile qualifier in the typedef definition, as in:

```
typedef uint32_t volatile device_register;
```

I've often seen C headers that define symbols for device register addresses as clusters of related macros. For example:

```
// timer registers
#define TMOD  ((unsigned volatile *)0xFFFF6000)
#define TDATA ((unsigned volatile *)0xFFFF6004)
#define TCNT  ((unsigned volatile *)0xFFFF6008)
```

defines TMOD, TDATA, and TCNT as the addresses of the timer mode register, the timer data register, and the timer count register, respectively. The header might also include useful constants for manipulating the registers, such as:

```
#define TE 0x01
#define TICKS_PER_SEC 50000000
```

which defines TE as a mask for setting and clearing the timer enable bit in the TMOD register, and TICKS_PER_SEC as the number of times the TCNT register decrements in one second.

Using these definitions, you can disable the timer using an expression such as:

```
*TMOD &= ~TE;
```

or prepare the timer to count for two seconds using:

```
*TDATA = 2 * TICKS_PER_SEC;
*TCNT = 0;
```

Other devices require similar clusters of macros, such as **Listing 1**. In this case, UART0 and UART1 have identical sets of registers, but at different memory-mapped addresses. They can share a common set of bit masks:

```
#define RDR 0x20
#define TBE 0x40
```

SO WHAT'S NOT TO LIKE?

This approach leaves a lot to be desired. As Scott Meyers likes to say, interfaces should be "easy to use correctly and hard to use incorrectly."[8] Well, this scheme leads to just the opposite.

By itself, disabling a timer isn't a hard thing to do. When you're putting together a system with thousands of lines of code controlling dozens of devices, writing:

```
*TMOD &= ~TE;
```

is asking for trouble. You could easily write |= instead of &=, leave off the ~, or select the wrong mask. Even after you get everything right, the code is far from self-explanatory.

Rather, you should package this operation as a function named timer_disable, or something like that, so that you and your cohorts can disable the timer with a simple function call. The function body is very short—it's only a single assignment—so you should probably declare it as an inline function. If you're stuck using an older C dialect, you can make it a function-like macro.

But what should you pass to that call? Most devices have several registers. Is it really a good idea to make the caller responsible for selecting which of the timer registers to pass to the function, as in:

```
timer_disable(TMOD);
```

when only one register will do? Maybe it's better to just build all the knowledge into the function, as in:

```
inline
void timer_disable(void)
     {
     *TMOD &= ~TE;
     }
```

so that all you have to do is call:

```
timer_disable();
```

This works for the timer because there's only one timer. However, my sample machine has two UARTs, each with six registers, and many UART operations employ more than one register.

For example, to send data out a port, you must use both the UART status register and the transmit buffer register. The function call might look like:

Listing 1

```
// UART 0 registers
#define ULCON0  ((unsigned volatile *)0xFFFFD000)
~~~
#define USTAT0  ((unsigned volatile *)0xFFFFD008)
#define UTXBUF0 ((unsigned volatile *)0xFFFFD00C)
~~~

// UART 1 registers
#define ULCON1  ((unsigned volatile *)0xFFFFE000)
~~~
#define USTAT1  ((unsigned volatile *)0xFFFFE008)
#define UTXBUF1 ((unsigned volatile *)0xFFFFE00C)
~~~
```

```
UART_put(USTAT0, UTXBUF0, c);
```

Maybe passing two registers isn't that bad, but what about operations that require three or even four registers?

Beyond the inconvenience, calling functions that require multiple registers invites you to make mistakes such as:

```
UART_put(USTAT0, UTXBUF1, c);
```

which passes registers from two different UARTs. In fact, the function even lets you accidentally pass a timer register to a UART operation, as in:

```
UART_put(USTAT, TDATA, c);
```

Ideally, compilers should catch these errors, but they can't. The problem is that, although each macro has a different value, they all yield expressions of the same type, namely, "pointer to volatile unsigned." Consequently, compilers can't use type checking to tell them apart.

Using different typedefs doesn't help. A typedef doesn't define a new type; it's just an alias for some other type. Thus, even if you define the registers as:

```
// timer registers
typedef uint32_t volatile timer_register;

#define TMOD    ((timer_register *)0xFFFF6000)
#define TDATA   ((timer_register *)0xFFFF6004)
~~~

// UART 0 registers
typedef uint32_t volatile UART_register;

#define ULCON0  ((UART_register *)0xFFFFD000)
#define UCON0   ((UART_register *)0xFFFFD004)
~~~
```

then `timer_register` and `UART_register` are just two different names for the same type, and you can use them interchangeably throughout your code. Even if you take pains to declare the `UART_put` function as:

```
void UART_put(UART_register *s,
    UART_register *b, int c);
```

you can still pass it a timer register as a UART register. By any name, an volatile unsigned is still an volatile unsigned.

Again, you could write the UART functions so that they know which registers to use. But there are two UARTs, so you'd have to write pairs of nearly identical functions, such as:

```
void UART0_put(int c);
void UART1_put(int c);
```

This gets tedious quickly, and becomes prohibitive when the number of UARTs is much bigger than two.

As an alternative, you could pass an integer designating a UART, as in:

```
typedef unsigned UART_number;

void UART_put(UART_number n, int c);
```

To make this work, you need a scheme that converts integers into register addresses at run time. Plus, you have to worry about what happens when you pass an integer that's out of range.

USING STRUCTURES

Structures provide a better way to model memory-mapped devices. You can use a structure to represent each collection of device registers as a distinct type. For example:

```
typedef uint32_t volatile device_register;

typedef struct timer_registers timer_registers;
struct timer_registers
    {
    device_register TMOD;
    device_register TDATA;
    device_register TCNT;
    };
```

The `typedef` before the struct definition elevates the tag name `timer_registers` from a mere tag to a full-fledged type name.[9] It lets you refer to the struct type as just `timer_registers` rather than as `struct timer_registers`.

You can provide corresponding structures for each device type:

```
typedef struct UART_registers UART_registers;
struct UART_registers
    {
    device_register ULCON;
    device_register UCON;
    device_register USTAT;
    device_register UTXBUF;
    ~~~
    };
```

Using these structures, you can define pointers that let you access device registers. You can define the pointers as macros:

```
#define the_timer
    ((timer_registers *)0xFFFF6000)
#define UART0 ((UART_registers *)0xFFFFD000)
```

THE EMBEDDED DEVELOPER
Microsoft
Windows Embedded
BUILT WITH WINDOWS 7 TECHNOLOGIES
Vol. 5
A MALFUNCTION IN THE SYSTEM COULD COST THE PLANT MILLIONS...
THE DEVICE HAS TO...
WORK PERFECTLY, TO THE MICROSECOND, AND HAVE THE CONNECTIVITY TO TRACK PERFORMANCE IN REAL TIME.
THEY'RE COUNTING ON ME TO DELIVER.
WINDOWS® EMBEDDED OFFERS A HIGHLY RELIABLE PLATFORM, WITH THE LEVEL OF PERFORMANCE YOU NEED TO HELP DELIVER CONNECTED DEVICES THAT STAND OUT.
WHICH WINDOWS® EMBEDDED-PLATFORM CAN HELP YOU DELIVER STANDOUT DEVICES?
FIND OUT AT WINDOWSEMBEDDED.COM/DEVICESTORIES

or as constant pointers:

```
timer_registers *const the_timer
     = (timer_registers *)0xFFFF6000;
UART_registers *const UART0
     = (UART_registers *)0xFFFFD000;
```

In C++, using a `reinterpret_cast` is even better:

```
timer_registers *const the_timer
     = reinterpret_cast<timer_registers *>
          (0xFFFF6000);
UART_registers *const UART0
     = reinterpret_cast<UART_registers *>
          (0xFFFFD000);
```

Whichever way you define the pointers, you can use them to access the actual device registers.

For example, you can disable the timer using the expression:

```
the_timer->TMOD &= ~TE;
```

Even better, you can wrap it in an inline function:

```
inline
void timer_disable(timer_registers *t)
     {
     t->TMOD &= ~TE;
     }
```

or a function-like macro:

```
#define timer_disable(t) ((t)->TMOD &= ~TE)
```

Whether you use an inline function or a macro, you can simply call:

```
timer_disable(the_timer);
```

For device operations that use more than one register, you can pass just the address of the entire register collection rather than individual registers. Again, sending data to a UART uses both the UART status register and the transmit buffer register. You can declare the `UART_put` function as:

```
void UART_put(UART_registers *u, int c);
```

and write it so that it picks out the specific registers that it needs. A call to the function looks like:

```
UART_put(UART0, c);
```

which is just a tad simpler than it was before. Plus, you can't accidentally mix registers from two UARTs at once.

Using structures avoid other mistakes as well. Each struct is a truly distinct type. You can't accidentally convert a "pointer to `timer_registers`" into a "pointer to `UART_registers`." You can only do it intentionally using a cast. Thus, compilers can easily catch accidents such as:

```
timer_disable(UART0);        // compile error
UART_put(the_timer, c);      // compile error
```

One of the problems with using structures to model collections of memory-mapped registers is that compilers have some freedom to insert unused bytes, called *padding*, after structure members.[10] You may have to use compile switches or `pragma` directives to get your structures just so.[4] You can also use compile-time assertions to verify that the structure members are laid as they should be.[11]

! For device operations that use more than one register, you can pass just the address of the entire register collection rather than individual registers.

CLASSES ARE EVEN BETTER

In C++, using a class to model hardware registers is even better than using a struct. I'll show you why over the coming months. ■

ENDNOTES:

1. Saks, Dan. "Mapping Memory," *Embedded Systems Programming*, September 2004, p. 49. *www.embedded.com/26807176*
2. Saks, Dan. "Mapping Memory Efficiently," *Embedded Systems Programming*, November 2004, p. 47. *www.embedded.com/50900224*
3. Saks, Dan. "More ways to map memory," *Embedded Systems Programming*, January 2005, p. 7. *www.embedded.com/55301821*
4. Saks, Dan. "Sizing and Aligning Device Registers," *Embedded Systems Programming*, May 2005, p. 9. *www.embedded.com/55301821*
5. Barr, Michael. "Introduction to fixed-width integers," *Embedded.com*, January 2004. *www.embedded.com/17300092*
6. Saks, Dan."Use Volatile Judiciously," *Embedded Systems Programming*, September 2005, p. 8. *www.embedded.com/170701302*
7. Saks, Dan."Place Volatile Accurately," *Embedded Systems Programming*, November 2005, p. 11. *www.embedded.com/174300478*
8. Meyers, Scott. "The Most Important Design Guideline?" *IEEE Software*, July/August 2004, p.14. *www.aristeia.com/Papers/IEEE_Software_JulAug_2004_revised.htm*
9. Saks, Dan. "Tag Names vs. Type Names," *Embedded Systems Programming*, September 2002, p. 7. *www.embedded.com/9900748*
10. Saks, Dan. "Padding and rearranging structure members," *Embedded Systems Design*, May 2009, p. 11. *www.embedded.com/217200828*
11. Saks, Dan, "Catching Errors Early with Compile-Time Assertions," *Embedded Systems Programming*, July 2005, p. 7. *www.embedded.com/columns/164900888*

Tektronix®

power2solve
SEMINAR TOUR

Helping engineers solve complex debug challenges quickly

You're invited.

Power2Solve is a series of FREE technical seminars for Design Engineers and Educators, offering practical, how-to training on the latest troubleshooting techniques for embedded system design. You'll spend a day working side-by-side with your peers and industry experts from Tektronix finding solutions to your debugging challenges.

Courses Include:

- Power Analysis
- Debugging Digital Designs
- Serial and Parallel Data Debug
- Characterization & Jitter Analysis

The Power2Solve Seminar is coming to a city near you:

May 18, 2010	The Hyatt Regency	Richardson,TX
May 20, 2010	The Radisson Hotel	Denver/Longmont, CO
May 24, 2010	The Hyatt Regency	San Diego, CA
May 26, 2010	The Hyatt Regency	Santa Clara, CA

ENTEST Instrument Engineers TECHNI-TOOL TESTEQUITY

Hands-on Training

Use Tektronix cutting-edge oscilloscopes to perform hands-on digital debug and serial data measurements.

Attend the event and get the opportunity to win the new **Apple iPad!**

Limited Seating — Register Now.
www.tektronix.com/power2solve

Applying secure programming standards and methodology can reduce vulnerabilities in software.

Bullet-proofing your software design

BY NAT HILLARY

In August 2003, a rolling blackout affected 10 million people in Ontario and 45 million people in the eastern part of the United States, raising concern that a cyber-attack by a hostile force was underway. Ultimately the causes of the blackout were traced to a slew of system, procedural, and human errors and not an act of aggression. Nevertheless, the event brought home the vulnerability of critical infrastructures connected to the Internet, raising awareness of the need for secure system components that are immune to cyber attack.

This increasing dependence on Internet connectivity demonstrates the growing need to build security into software to protect against currently known and future vulnerabilities. This article will look specifically at the best practices, knowledge, and tools available for building secure software that's free from vulnerabilities.

SECURE SOFTWARE

In his book *The CERT C Secure Coding Standard*, Robert Seacord points out that there is currently no consensus on a definition for the term *software security*. For the purposes of this article, the definition of *secure software* will follow that provided by the U.S. Department of Homeland Security (DHS) Software Assurance initiative in "Enhancing the Development Life Cycle to Produce Secure Software: A Reference Guidebook on Software Assurance." DHS maintains that software, to be considered secure, must exhibit three properties:

1. ***Dependability***—Software that executes predictably and operates correctly under all conditions.
2. ***Trustworthiness***—Software that contains few, if any, exploitable vulnerabilities or weaknesses that can

#define USTATO ((unsigned volatile *
#define ULCON1 ((unsigned
ned volatile *)0xFFFFE0

be used to subvert or sabotage the software's dependability.

3. *Survivability* (also referred to as "Resilience")—Software that is resilient enough to withstand attack and to recover as quickly as possible, and with as little damage as possible from those attacks that it can neither resist nor tolerate.

The sources of software vulnerabilities are many, including coding errors, configuration errors, and architectural and design flaws. However, most vulnerabilities result from coding errors. In a 2004 review of the National Vulnerabilities Database for their paper "Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?" to the 37th International Conference on System Sciences, Jon Heffley and Pascal Meunier found that 64% of the vulnerabilities resulted from programming errors. Given this, it makes sense that the primary objective when writing secure software must be to build security in.

BUILDING SECURITY IN

Most software development focuses on building high-quality software, but high-quality software is not necessarily secure software. Consider the office, media-playing, or web-browsing software that we all use daily; a quick review of the Mitre Corporation's Common Vulnerabilities and Exposures (CVE) dictionary will reveal that vulnerabilities in these applications are discovered and reported on an almost weekly basis. The reason is that these applications were written to satisfy functional, not security requirements. Testing is used to verify that the software meets each requirement, but security problems can persist even when the functional requirements are satisfied. Indeed, software weaknesses often occur by the unintended functionality of the system.

Adding a security perspective to software requirements ensures that security is included in the definition of system correctness.

Building secure software requires adding security concepts to the quality-focused software-development lifecycle so that security is considered a quality attribute of the software under development. Building secure code is all about eliminating known weaknesses (**Figure 1**), including defects, so by necessity secure software is high-quality software.

Security must be addressed at all phases of the software development lifecycle, and team members need a common understanding of the security goals for the project and the approach that will be taken to do the work.

The starting point is an understanding of the security risks associated with the domain of the software under development. This is determined by a security risk assessment, a process that ensures the nature and impact of a security breach are assessed prior to deployment in order to identify the security controls necessary to mitigate any identified impact. The identified security controls then become a system requirement.

Adding a security perspective to software requirements ensures that security is included in the definition of system correctness that then permeates the development process. A specific security requirement might validate all user string inputs to ensure that they do not exceed a maximum string length. A more general one might be to withstand a denial of service attack. Whichever end of the spectrum is used, it is crucial that the evaluation criteria are identified for an implementation.

When translating requirements into design, it is prudent to consider security risk mitigation via architectural design. This can be in the choice of implementing technologies or by inclusion of security-oriented features, such as handling untrusted user interactions by validating inputs and/or the system responses by an independent process before they are passed on to the core processes.

The most significant impact on building secure code is the adoption of secure coding practices, including both static and dynamic assurance measures. The biggest bang for the buck stems from the enforcement of secure coding rules via static analysis tools. With the introduction of security concepts into the requirements process, dynamic assurance via security-focused testing is then used to verify that security features have been implemented correctly.

CREATING SECURE CODE WITH STATIC ANALYSIS

A review of the contents of the CVE dictionary reveals that common software defects are the leading cause of security vulnerabilities. Fortunately, these

Building secure code by eliminating known weaknesses.

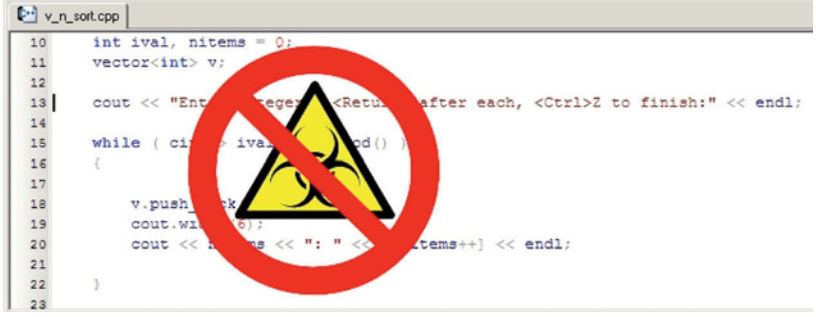
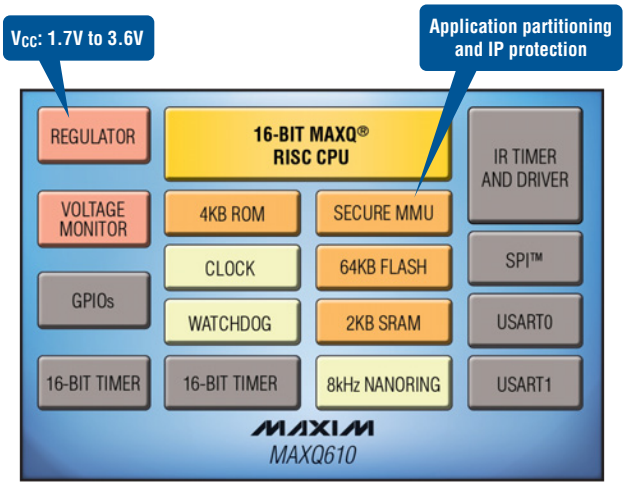


Figure 1

Extend battery life with the MAXQ610 16-bit microcontroller

Up to 12 MIPS in under 4mA—industry's highest MIPS/mA!

The MAXQ610 microcontroller is designed for low-cost, high-performance, battery-powered applications. This 16-bit, RISC-based microcontroller has a wide operating range (down to 1.7V) for long battery life and ultra-low power consumption. Its anticloning features and secure MMU enable you to protect your IP.



Microcontroller

- 16-bit MAXQ RISC core
- 64KB flash memory, 2KB SRAM
- Ultra-low supply current
  - Active mode: 3.75mA at 12MHz
  - Stop mode: 200nA (typ), 2.0µA (max)
- Wide, 1.7V to 3.6V operating voltage range
- IP protection
  - Secure MMU supports multiple privilege levels, helps protect code from copying and reverse engineering

Peripherals

- Two USARTs and one SPI master/slave communication port
- Two 16-bit timers/counters
- 8kHz nanoring functions as programmable wakeup timer
- IR timer simplifies support for low-speed infrared communication
- IR driver capable of 25mA sink current

Part	Temp Range (°C)	Program Memory	Data Memory	Operating Voltage (V)	Package	Price† ($)
MAXQ610B	0 to 70	64KB flash	2KB SRAM	1.7 to 3.6	40-TQFN	1.45

MAXQ is a registered trademark of Maxim Integrated Products, Inc.
SPI is a trademark of Motorola, Inc.
†1000-up recommended resale. Prices provided are for design guidance and are FOB USA. International prices will differ due to local duties, taxes, and exchange rates. Not all packages are offered in 1k increments, and some may require minimum order quantities.

www.maxim-ic.com/MAXQ610-info

MAXIM DIRECT
www.maxim-ic.com/shop

AVNET electronics marketing
www.em.avnet.com/maxim

MAXIM INNOVATION DELIVERED™

For free samples or technical support, visit our website.

Innovation Delivered is a trademark and Maxim is a registered trademark of Maxim Integrated Products, Inc. © 2010 Maxim Integrated Products, Inc. All rights reserved.

vulnerabilities can be attributed to common weaknesses in code, and a number of dictionaries have been created to capture this information, such as the Common Weakness Enumeration (CWE) dictionary from the Mitre Corporation and the CERT-C Secure Coding Standard from the Software Engineering Institute at Carnegie Mellon. These secure coding standards can be enforced by the use of static analysis tools, so that even novice secure software developers can benefit from the experience and knowledge encapsulated within the standards.

The use of coding standards to eliminate ambiguities and weaknesses in the code under development has been proven extremely successful in the creation of high-reliability software, such as the use of the Motor Industry Software Reliability Association (MISRA) Guidelines for the use of the C language in critical systems. The same practice can be used to similar effect in the creation of secure software.

Of the common exploitable software vulnerabilities that appear in the CVE dictionary, some occur more than others—user input validation, buffer overflows, improper data types, and improper use of error and exception handling. The CWE and CERT-C dictionaries identify coding weaknesses that can lead to these vulnerabilities. The standards in each of these dictionaries can be enforced by the use of static analysis tools that help to eliminate both known and unknown vulnerabilities while also eliminating latent errors in code. For example, the screenshot in **Figure 2** shows the detection of a buffer overflow vulnerability due to improper data types.

Static software analysis tools assess the code under analysis without executing it and are adept at identifying coding standard violations.

Static software analysis tools assess the code under analysis without actually executing it. They are particularly adept at identifying coding standard violations. In addition, they can provide a range of metrics that can be used to assess and improve the quality of the code under development, such as the cyclomatic complexity metric that identifies unnecessarily complex software that's difficult to test.

When using static analysis tools for building secure software, the primary objective is to identify potential vulnerabilities in code. Example errors that static analysis tools identify include:

- Insecure functions
- Array overflows
- Array underflows
- Incorrectly used signed and unsigned data types

Since secure code must, by nature, be high-quality code, static analysis tools can be used to bolster the quality of the code under development. The objective here is to ensure that the software under development is easy to verify. The typical validation and verification phase of a project can take up to 60% of the total effort, while coding typically only takes 10%. Eliminating defects via a small increase in the coding effort can significantly reduce the burden of verification, and this is where static analysis can really help.

Ensuring that code never exceeds a maximum complexity value helps to enforce the testability of the code. In addition, static analysis tools identify other issues that affect testability, such as having unreachable or infeasible

code paths or an excessive number of loops.

By eliminating security vulnerabilities, identifying latent errors, and ensuring the testability of the code under development, static analysis tools help ensure that the code is of the highest quality and secure against not only current threats but unknown threats as well.

FITTING TOOLS INTO THE PROCESS

Tools that automate the process of static analysis and enforcement of coding standards such as CWE or CERT C Secure Coding guidelines ensure that a higher percentage of errors are identified in less time. This rigor is complimented by additional tools for:

- *Requirements traceability*—a good requirements traceability tool is invaluable to the build security in process. Being able to trace requirements from their source through all of the development phases and down to the verification activities and artifacts ensures the highest quality, secure software.
- *Unit testing*—the most effective and cheapest way of ensuring that the code under development meets its security requirements is via unit testing. Creating and maintaining the test cases required for this, however, can be an onerous task. Unit testing tools that assist in the test-case generation, execution and maintenance streamline the unit testing process, easing the unit testing burden and reinforcing unit test accuracy and completeness.
- *Dynamic analysis*—analyses performed while the code is executing provide valuable insight into the code under analysis that goes beyond test-case execution. Structural coverage analysis, one of the more popular dynamic analysis methods, has been proven to be invaluable for ensuring that the verification test cases execute all of the code under development. This helps ensure that there are no hidden vulnerabilities or defects in the code under development.

The most effective and cheapest way of ensuring that the code under development meets its security requirements is via unit testing.

While these various capabilities can be pieced together from a number of suppliers, some companies offer an integrated tool suite that facilitates the building security in process, providing

LDRA TBvision screenshot showing improper data type sign usage resulting in buffer overflow vulnerability.

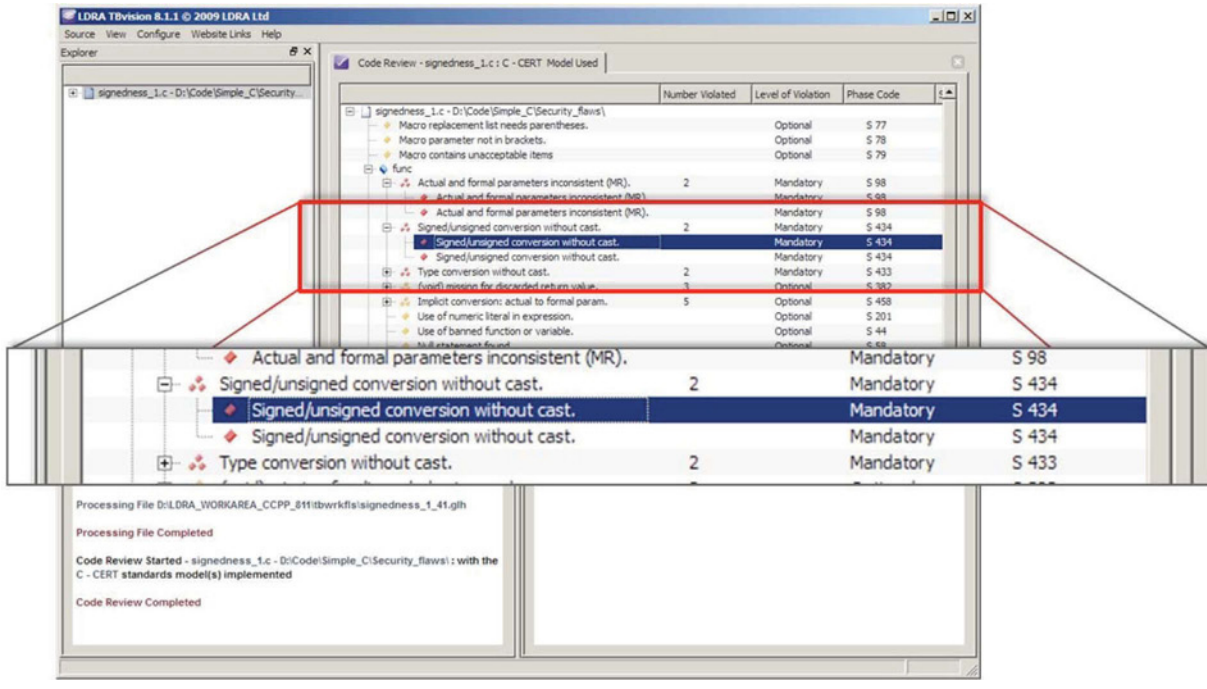

LDRA TBvision 8.1.1 © 2009 LDRA Ltd
Source View Configure Website Links Help
Explorer
Code Review - signedness_1.c : C - CERT Model Used
Number Violated
Level of Violation
Phase Code
Actual and formal parameters inconsistent (MR). Mandatory S 98
Signed/unsigned conversion without cast. 2 Mandatory S 434
Signed/unsigned conversion without cast. Mandatory S 434
Signed/unsigned conversion without cast. Mandatory S 434
Type conversion without cast. 2 Mandatory S 433
Processing File D:\LDRA_WORKAREA_CCPP_811\tbwrkfls\signedness_1_41.glh
Processing File Completed
Code Review Started - signedness_1.c - D:\Code\Simple_C\Security_flaws\ : with the C - CERT standards model(s) implemented
Code Review Completed

Figure 2

SEGGER

Emulator (J-Link™)

ARM and Cortex

- USB to JTAG Emulator
- Fast 720kb/s Download Speed
- Serial Wire Debug (SWD) Support
- Multicore Debugging Support
- Auto JTAG Speed Recognition

8.12.00 J-Link ARM PRO

Adds Ethernet Conectivity and Licensing for All Enhancement Modules

The J-Link can be coupled with a number of available software modules to fit your application needs.

J-Flash is a stand-alone application used with the J-Link to program internal and external flash devices.

J-Link RDI permits the use of the J-Link with an RDI compliant debugger.

J-Link GDB Server is a remote server for GDB.

J-Link Flash Breakpoint permits you to set an unlimited number of software breakpoints while debugging in flash.

As the original manufacturer of the J-Link and its OEM derivatives, we are happy to inform you that this software also supports the DIGI® JTAG Link, Atmel® SAM-ICE, and IAR® J-Link KS.

$60

Special Offer

8.08.90 J-Link EDU
Educational Use J-Link
Includes Flash Breakpoints
Includes J-Link GDB Server
www.segger-us.com/edu.html

J-Link Educational Use (EDU) Bundle

www.segger.com

Secure coding in the iterative lifecycle.

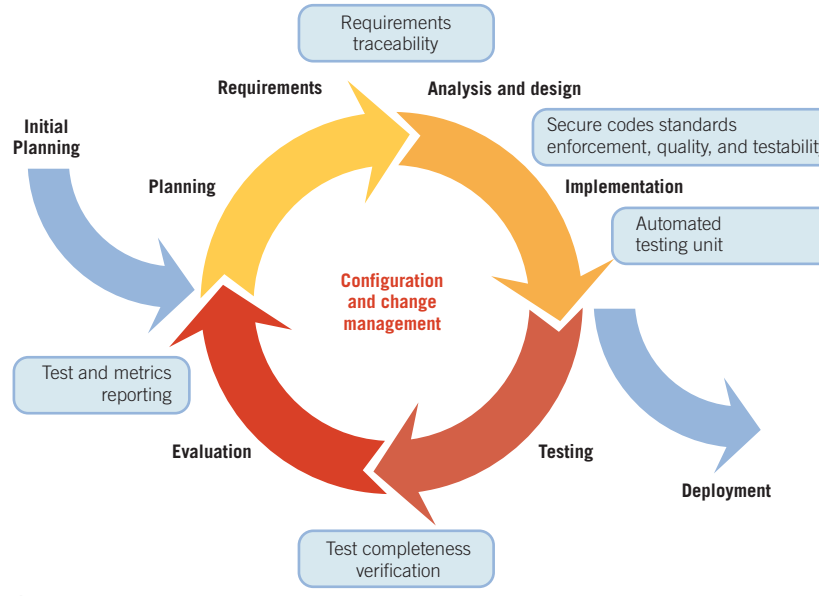


Figure 3

all of the solutions described above.

> **It's not surprising that the processes for building security into software echoes the high-level processes required for building quality into software.**

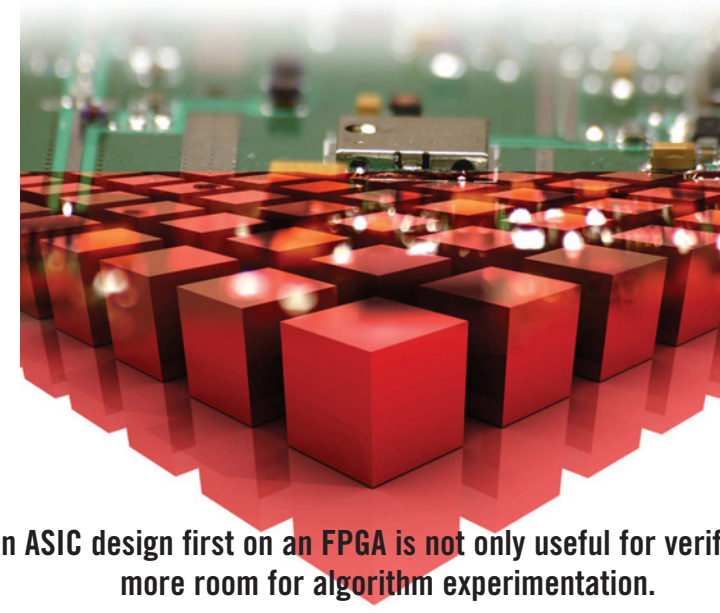
BUILDING SECURITY

It's not surprising that the processes for building security into software echoes the high-level processes required for building quality into software. Adding security considerations into the process from the requirements phase onwards is the best way of ensuring the development of secure code, as described in **Figure 3**. High-quality code is not necessarily secure code, but secure code is always high-quality code.

An increased dependence on internet connectivity is driving the demand for more secure software. With the bulk of vulnerabilities being attributable to coding errors, reducing or eliminating exploitable software security weaknesses in new products through the adoption of secure development practices should be achievable within our lifetime.

By leveraging the knowledge and experience encapsulated within the CERT-C Secure Coding Guidelines and CWE dictionary, static analysis tools help make this objective both practical and cost effective. Combine this with the improved productivity and accuracy of requirements traceability, unit testing and dynamic analysis and the elimination of exploitable software weaknesses become inevitable. ■

Nat Hillary is a field applications engineer with LDRA Technologies, Inc., a position that he comes to via an extensive background in software engineering, sales and marketing. He is an experienced presenter in the use of software analysis solutions for real-time safety critical software who has been invited to participate in a number of international forums and seminars on the topic.



Prototyping an ASIC design first on an FPGA is not only useful for verification but allows more room for algorithm experimentation.

How to use an FPGA to test a PLL band calibration algorithm

BY RUSSELL MOHN

It's a common technique to split the required frequency tuning range of a controlled oscillator into discrete bands. The advantage of having many bands is that a wide tuning range can be covered while keeping a relatively low voltage-controlled oscillator (VCO) gain within each band. Low VCO gain is good for achieving low VCO phase noise. It's required that the frequency bands overlap. The tuning bands are changed with a digital band control signal.

When an oscillator with discrete tuning bands is used in a phase-locked loop (PLL), the desired band must be selected before the PLL can proceed to phase lock. This necessary step has many names (band calibration, auto-band selection, band selection, and so on), but the idea is the same: to pick the right frequency band before allowing the PLL to lock.

A straightforward way to calibrate the band is by racing two counters, one clocked with the reference clock and the other clocked with the feedback clock that is the frequency divided version of the VCO output. The frequency division occurs in a block called a multi-modulus divider (MMD).

The counters are forced to start at the same time and permitted to count up to a predetermined value. Whichever counter gets to the value first is noted as the winner; it follows that that clock was greater in frequency.

Using the information about which counter was the winner, the band control of the VCO can be either incremented or decremented to bring the frequencies closer. This algorithm is implemented in a band calibration block (BCAL). Instead of waiting for an expensive ASIC fabrication run that includes the entire PLL and other circuits, you can implement a band calibration algorithm and test it on an FPGA. This article shows you how.

VCO BAND CALIBRATION (BCAL)

In communications chips, frequency synthesizers are ubiquitous functional blocks. A frequency synthesizer is loosely defined as a PLL that generates an output frequency that's directly proportional to a reference frequency. The constant of proportionality is a specific subset of integer or real numbers, depending on the synthesizer implementation.

One use for a synthesizer in a receiver front-end is the creation of the local oscillator input to a mixer that downconverts the received radio frequency (RF) signal to an intermediate frequency. Channel selection is achieved by setting the synthesizer's constant of proportionality. In general, RF = Ndiv * REF, where RF is the output frequency, Ndiv is the constant of proportionality, and REF is the reference frequency.

Ndiv can be a ratio of integers, N/R, where N is an integer divide value for the output of the VCO, and R is another integer divide ratio for dividing the reference oscillator. If even finer frequency resolution is needed, the N value can be added to a sigma-delta modulated code that dithers the divider function and gives a fractional resolution of REF/2^(# sigma-delta accumulator bits).

A large VCO gain would make the PLL susceptible to high phase noise. For these reasons, the tuning range is split up into discrete bands.

Frequency synthesizers multiply a fixed frequency crystal oscillator up to the required frequency. The PLL acts as a closed-loop negative feedback system to implement this exact multiplication. The job of the MMD is to divide the frequency of the VCO output by the integer value N.

The phase of this signal is compared with the phase of the reference, and the difference in phases is filtered to remove high-frequency components. The filtered signal is used as the voltage control of the VCO. If there is any phase difference between the output of the MMD and the reference, the control voltage at the VCO will adjust to correct that phase difference.

For the application at hand, the synthesizer needed to create frequencies from 3,000 to 4,000 MHz. Continuous tuning of the VCO is accomplished by changing the bias voltage across a varactor which is part of the parallel inductor-capacitor (LC) resonant circuit. The fabrication technology limits the control voltage to a maximum change of about 1.5 V. It's difficult to build a varactor that will change its reactance enough to cause a frequency change of 1,000 MHz with only a control voltage change of 1.5 V.

Furthermore, a large VCO gain of 1,000 MHz/1.5 V would make the PLL susceptible to high phase noise. For these reasons, the tuning range is split up into discrete bands. The discrete bands are implemented by adding binary-weighted capacitors to the parallel LC tank circuit. They are switched on or off depending on the digital band setting. The band must be set before the PLL can be allowed to lock and track in a continuous manner.

The BCAL circuit operates as a second feedback loop controlling the VCO through its band input. During band calibration, the VCO control voltage is fixed at a convenient voltage, usually the mid-point of its allowable control

The band calibration testbench implemented on the FPGA is analogous to band calibration used in a frequency synthesizer on an ASIC.

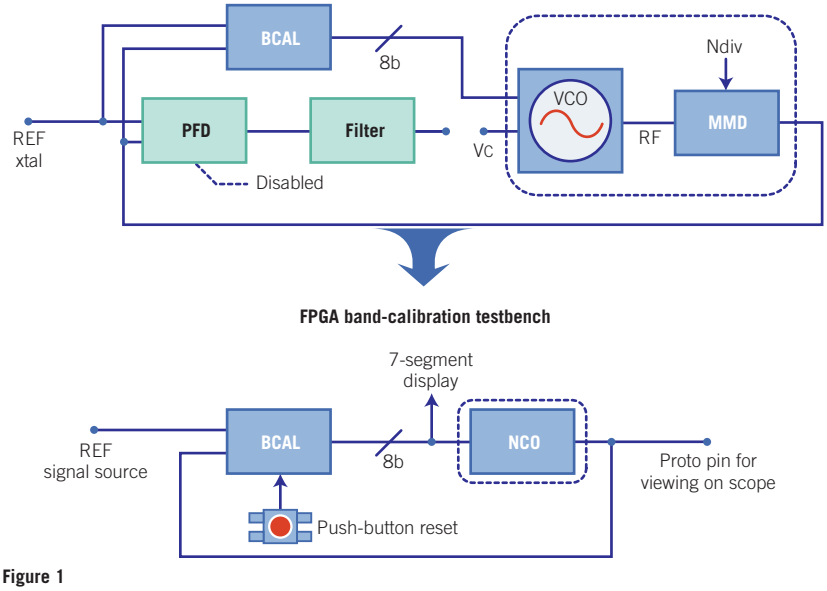


Figure 1

voltage range. The phase-detector is also disabled during band calibration.

My goal was to design and test the band calibration algorithm before integrating it with the PLL on an RF receiver ASIC. To that end, a system analogous to the PLL when it's being band-calibrated was constructed entirely with circuits that could be implemented on an FPGA. Since the VCO and MMD lumped together act as a programmable oscillator with output frequencies around the reference frequency, their functionality can be modeled by a numerically-controlled oscillator (NCO), shown in **Figure 1**.

For the synthesizer to have low phase noise, a crystal generates the frequency reference. The reference frequency is typically in the tens of MHz, which is well below the maximum speed of the logic that can be implemented on today's FPGAs. The BCAL algorithm itself can be described and designed with digital techniques.

At its simplest, its inputs are two clocks, the reference and the output of the NCO; its output is the band signal for the NCO. The combination of the band calibration, NCO, and an externally applied reference signal forms a closed loop system with negative feedback that is analogous to the PLL operating during its band calibration mode, all of which can be coded in RTL and tested on an FPGA before spending money on an ASIC fabrication.

WHAT YOU NEED

1. An FPGA and its programming software
2. Matlab/Simulink for algorithm development and verification
3. A signal source for generating the reference clock, such as 10 to 15 MHz
4. An oscilloscope for debugging

I used Matlab/Simulink to enter the initial design and testbench. The support for fixed-point numbers that comes with the Fixed-Point Toolbox and Simulink Fixed Point is useful for making the model accurately reflect the implementation in RTL. The RTL code was written in verilog and run on Altera's Stratix II DSP Development Kit.

From within Altera's Quartus II software all-things-FPGA could be accomplished: design entry, simulation for functionality, simulation for timing, synthesis, fitting, configuring the FPGA with the design, and debugging. When I tested the band-calibration in real time, I used the signal source and oscilloscope.

The binary search block holds the current band output value and determines what the next band value will be based on which clock won the race.

DESIGN AND PROTOTYPING PROCEDURE

The design and prototyping procedure is the iteration of the following familiar steps: 1. Design Entry; 2. Test; 3. Debug; 4. Go To 2. This cycle is repeated as many times as necessary until the desired functionality is reached.

First, I built the NCO as a Simulink subsystem. The NCO Simulink model was reverse-engineered from the verilog for an NCO I found on the web at *www.mindspring.com/~tcoonan/nco.v*. The NCO was based on a programmable modulo counter. Its output frequency equals Fs*(BAND+STEP)/MOD where STEP and MOD are fixed values and BAND is the 8-bit band signal.

The NCO's functionality was verified by running transient simulations using Fs=11MHz and sweeping through the BAND values, 0 to 255, and calculating the resulting output frequency. The resulting output frequency versus BAND, or band tuning curve, was monotonic but not perfectly linear. Since it was monotonic, it was deemed acceptable to use in the closed-loop test setup for the BCAL.

After establishing that the NCO has a monotonic tuning curve and can produce frequencies in the range 10 to 14 MHz, which is approximately the PLL's reference frequency, I built the BCAL model. The BCAL algorithm works by racing two identical 10-bit counters. One counter is clocked by the reference; the NCO clocks the other.

Since they both start from 0, the first counter to get to a constant HIT_VALUE, is clocked by the greater frequency. To determine which counter gets to HIT_VALUE first, each count value is continuously compared with the HIT_VALUE, and the XOR of the two comparison results is used to clock a "1" into a D flip-flop.

When both count values are less than HIT_VALUE, the comparators both output 0, and the XOR result is 0. At the instant one of the values exceeds the HIT_VALUE, the XOR output transitions to 1 and captures a 1 on the DFF output. Sometime thereafter, the other count value will get to HIT_VALUE, and the XOR result returns to 0.

Another comparator is used to compare the reference counter to a constant RESET_VALUE, and when the count exceeds this value, both counters are reset to 0 and the race begins over again. If the HIT_VALUE is 230, a plausible RESET_VALUE is 240. Meanwhile, the bit of information about which clock was faster is used as input to a binary search block.

The binary search block holds the current band output value and determines what the next band value will be based on which clock won the race. The binary search block either adds or subtracts the appropriate binary weighted value from its current output. For an 8-bit band, the initial band value is mid-range at 128, and seven consecutive races are conducted to fill in the 8-bits from MSB to LSB. An example run of the BCAL algorithm is shown in **Figure 2**.

The BCAL algorithm finishes in 146 µs so only the final value appears to a human observer. The BCAL algorithm was pass/fail tested for 50 possible frequencies.

After building the band calibration algorithm in Simulink from logic gates, comparators, registers, delays, and look-up tables, the design was entered in the Quartus II software. To make debugging easier, every wire in the Simulink model was named.

During the translation process, I used the same names for signals in the Verilog code. If a signal originated from a register (or a delay in a triggered subsystem) in the Simulink model, I made it a register in Verilog; otherwise the signal was a wire. As a result, the design entry from Simulink primitive subsystems to Verilog was straightforward.

In a manner similar to the testing done in Simulink, all the submodules were simulated and verified in Quartus II. After functionality was confirmed for the submodules, a test schematic for the entire BCAL was made. The test schematic includes the NCO which is controlled by the BCAL band output.

To complete the loop, the NCO output is used as one of the clock inputs to the BCAL. The BCAL reference input was wired through one of the FPGA pins to an SMA connector on the board so it could be clocked with an external signal source.

The BCAL testbench was synthesized and fitted, and the timing netlist was simulated. Immediately, it was apparent there was a bug in the design because some of the band bits were going into undefined states, shown as "U" in Quartus II.

The bug came from the asynchronous comparisons of the counter values to the HIT_VALUE. After registering these comparison results and retiming the asynchronous data paths to the reference clock, the design functionality was okay in simulation. The next step was to load the design onto the FPGA and verify through measurement.

The testing proceeded by changing the reference frequency generated by the signal source from 10 to 14 MHz in increments of approximately 100 kHz. The test setup is shown in **Figure 3**. At each reference frequency, the band calibration was initiated by a reset tied to a push-button. Switch debouncing would have made a cleaner testbench but was not necessary.

Multiple resets caused by the switch bounce cause the algorithm to start over repeatedly; when the switch stops bouncing, the BCAL operates normally. The 8-bit band value was mapped to two 7-segment displays on the FPGA board to display the final band value in hexadecimal.

The BCAL algorithm finishes in 146 µs (= 7*230/11 MHz), so only the final value appears to a human observer. The readout made it easy to compare against the theoretical value from the Simulink model. In this way, the BCAL algorithm was pass/fail tested for 50 possible frequencies from its minimum to maximum band values.

An example band calibration run with REF = 14.3MHz; the band settles to 227.

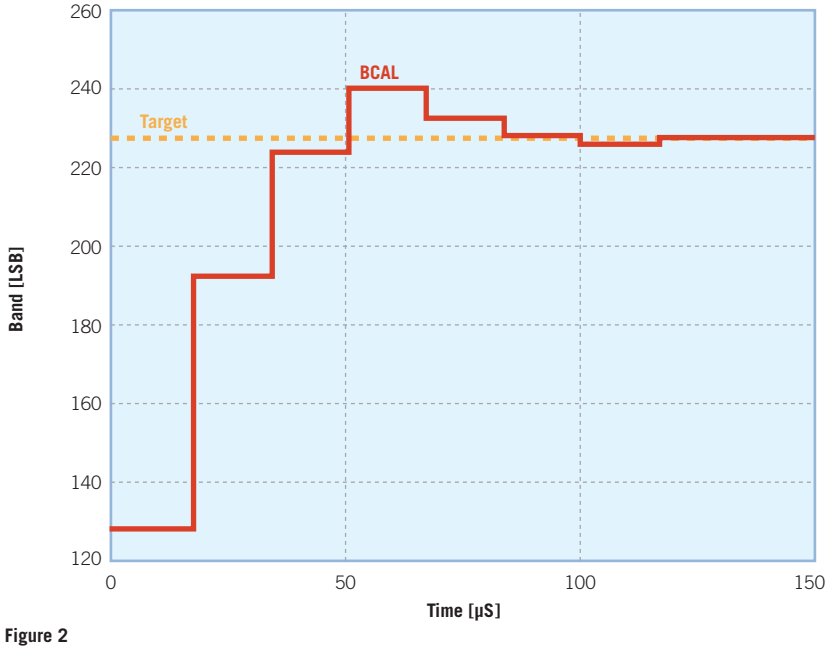


Figure 2

POTENTIAL PITFALLS AND TIPS

One of the challenges of this particular design was its asynchronous nature. The frequency of the NCO clock changes during the band calibration, and some logic elements in the BCAL depend on the timing of the edges of that clock. Likewise, other logic elements change synchronously to the reference clock edges.

The FPGA design software is not conducive to asynchronous design. It's not impossible to make an asynchronous design, but don't be surprised if you have to look through documentation on a collection of warnings to determine if your code does what you intend. Since the reference frequency never changes, the design was modified to make all the data paths synchronous to the reference clock.

When the data path needed to jump clock domains, it was retimed with cascaded registers to minimize metastability. Similarly, another pitfall was not registering combinatorial comparator outputs. These are both examples of problems that arise in actual hardware but may not show up in the idealized models in Simulink, unless you explicitly add them in your model.

To ease the migration of the Simulink model to RTL, try to use Simulink function blocks that are primitives in the RTL language of your choice. For example, logic functions such as XOR, AND, and greater-than map directly from Simulink to Verilog. A delay or explicit DFF in Simulink is modeled as a register in Verilog.

I also recommend naming all the signals in the Simulink model and using the same names in the Verilog code. It's okay to first build the model using floating-point data types in Simulink, but if you migrate the floating-point design to fixed-point, it will ease the coding process and lead to a design that is easier to debug.

THE END RESULT

After running the RTL code on the FPGA and judging that the design is functional and meets specifications based on measured data, it was time to implement the code on an ASIC. The logic synthesis and layout was done with Cadence's Encounter software. As a final check, I simulated the resulting logic netlist and also the extracted layout netlist with parasitic resistors and capacitors to make sure the functionality was still okay after Encounter's synthesis and place-and-route.

Lab setup for band calibration running on the FPGA showing that measured band value matches the simulation. Now, the RTL code can be implemented on the ASIC with confidence it will function correctly.

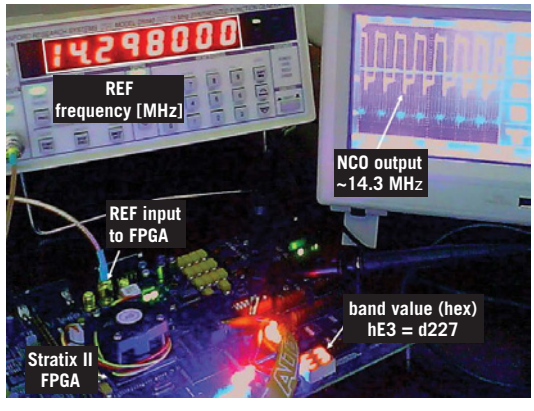


Figure 3

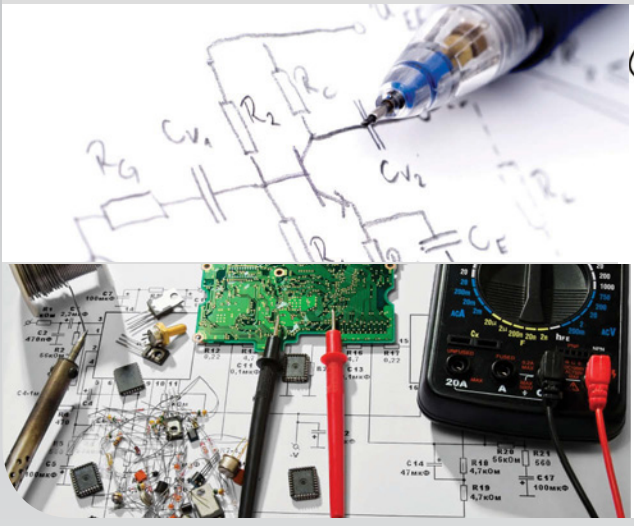
The functionality checked out okay in those simulations. Since then, the RF receiver ASIC that includes the frequency synthesizer was fabricated and measurements of the chip show the frequency synthesizer phase locks over its range of possible output frequencies. This implies that the band calibration functions correctly. As a result, the design team can focus on squeezing better performance out of the analog portions of the ASIC.

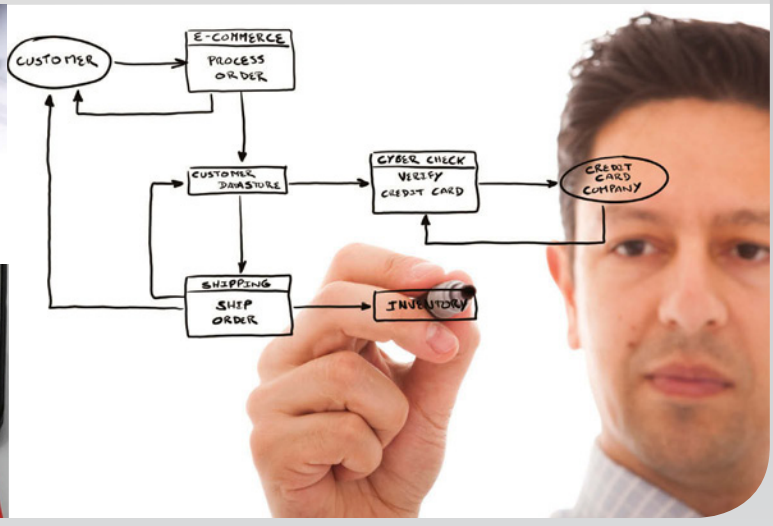
The process of prototyping a design on an FPGA before committing it to an ASIC is useful not only verification purposes but also for the possibilities it provides for algorithm experimentation. If the context of the algorithm can be replicated on the FPGA as it will appear on the ASIC, any number of algorithm implementations may be tried and compared in terms of area efficiency, current consumption, or speed. Happy prototyping! ■

Russell Mohn is a senior design engineer at Epoch Microelectronics, an IC design services company specializing in mixed-signal, analog, and RF design. At Epoch, his focus has been on the design of fractional-N frequency synthesizers for communications applications. In 2001, he received his B.E. in EE from Cooper Union. His interests include FPGA prototyping, system modeling, and signal processing.

Embedded & Network Computing Technologies
CALAO SYSTEMS
TNY-A9G20-LPW
Low Power 100mA@5V
Tiny Form Factor (36 x 41 mm)
Industrial Operating Temperature Range:
-40°C / +85°C
TNY-A9G20-LPW is designed for all projects with requirements for:
SMALL SIZE, HIGH PERFORMANCE & LOW POWER!
www.calao-systems.com

Join EE Times Group to learn how distributors stack up in today's market





EE Times **Live Webinar**

2010 Distributor Brand Preference Study

OEMs rank distributors based on how well they help them achieve their operating goals and today, more than ever, it's crucial for distributor partners to understand how they stack up in the face of their vendor partners and end customers, how they are measured and where they rate compared to their competitors in order to strengthen revenues and plan for a strong year ahead.

Tune in on May 14 as EE Times Group unveils the 2010 Distributor Brand Preference Study, our annual benchmark research on the $40 billion+ distribution market where design engineers, technical managers, supply chain management, purchasing and corporate managers across the OEM market have weighed in on:

- Which distributors are preferred and what criteria matters the most such as ease of doing business, customer service, pricing and website capabilities.
- How distributor brand preference varies across product categories such as semiconductors, connectors & interconnects, passive components and electromechanical devices
- What factors influence purchasing decisions from the same distributor

Presented by EE Times, this landmark research is a key measurement tool for OEM's, CEM's, distributors and suppliers. If you are a distributor, an OEM or a decision maker in the purchase process of electronic components you need to attend this webinar.

Registration information

- Register by 4/30/10: $250.00 (USD)
- Register between 5/1/10 and 5/14/10: $299.00 (USD)

The registration fee is payable by American Express, Mastercard, or Visa. All paid registrants will be given a confirmation number and a url which will enable them to view the study, at their leisure, from May 14 to December 31, 2010.

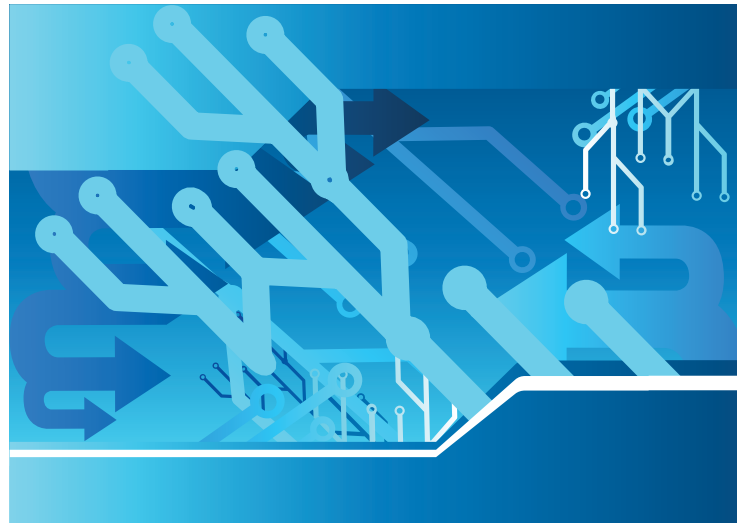
Please contact EE Times' Webinar Support with any questions at webinar@techonline.com

Presenter:

Jim McLeod-Warrick
Founding Partner of Beacon Technology Partners LLC

Register today: http://tinyurl.com/514-eet-study

EE Times Group



Here's a novel approach to the "update tool or update source" dilemma.

Dealing with misbehaving tools

BY ANDERS HOLMBERG

Making software changes very late in a project is almost never a good thing. Although correction of the error might be crucial for the correct and safe usage of the end product, it has a number of unwanted side effects:

- *The process view:* Making changes to the code and rebuilding the application image will force a restart of one or more test and quality assurance (QA) activities in the project. Minimizing test and QA without compromising safety and integrity in the face of code changes can thus become critical for time to market.

  The later in the process a problem is encountered the further back in the process you have to go to revisit certain activities. In the worst case, a full external revalidation or recertification assessment may be required.
- *The code view:* Changing code to correct erratic behavior always carries the risk of introducing new unwanted behavior. This sometimes leads to the decision to leave a problem as is in the product and docu-

ment clearly the impact of the code behavior. High-integrity regulatory frameworks often make things even tougher by requiring extensive impact analysis of the changes prior to performing them.

- *The goodwill view:* Frequent or large code changes in the final stages of a project can make stakeholders nervous about the end product. If the product has already reached the market, the situation can be a real nightmare.

So, it's not always possible to avoid code changes, but methods and tools to avoid or minimize the impact of changes can be extremely helpful.

Three broad categories cause most of the late code changes:

- *The source-code bug:* This kind of error is either due to mistakes or misunderstandings by the programmer in the implementation or an ambiguous or incomplete functional specification that leaves too much open to interpretation. Although this is a common occurrence, I won't be discussing it in this article.
- *The latent non-ANSI C/C++ source bug:* The ANSI C standard has some dark corners where behavior is either implementation defined or undefined. If you've implemented parts of the source code to depend on how a particular compiler behaves for these corner cases of the standard, you can expect a problem in the future. However, as this kind of latent bug is mainly a process and knowledge issue, it will not be discussed further here.
- This leaves us with the main focus for this article, ***the object-code-generation-tool bug.*** We will restrict the discussion to bugs in the build chain, in other words, the compiler, assembler, and linker.

It's not always possible to avoid code changes, but methods and tools to avoid or minimize the impact of changes can be extremely helpful.

Consider the following situation. A bug you have found is the result of the compiler making wrong assumptions about register allocation and stack allocation of variables that are local to a function. The bug is exposed when many variables compete for the available CPU registers and some variables have to be temporarily moved to the stack. You have found the bug in a large function with a lot of arithmetic computations, but that is no guarantee that the bug will only manifest itself in large functions with a lot of computations.

So we end up with the question of whether to persuade the compiler vendor to supply a fix or applying the workaround(s) throughout the code base with all the implications for the project outlined above.

For high integrity projects the build chain and the vendor should be subject to a lot of scrutiny before selection. And the typical scenario is that once a particular compiler and version is selected, you stay with it throughout the project. Some high-integrity process frameworks even require the tools selection to be subject to a formalized process where the tools are prequalified or validated according to certain criteria.

We will now take a look at a special technique that can be used if you have a close relationship with your compiler vendor. Consider a compiler for a 32-bit architecture. Many 32-bit CPU kernels incorporate some kind of instruction pipeline to increase performance by dividing complex instructions into 1-cycle pieces that are executed in their own pipeline stage.

In this way a throughput of one instruction per clock cycle can be achieved under ideal circumstances. It is however very easy to break this if subsequent instructions are competing for the same resource. An example is if the first instruction writes to a particular register and the directly following instruction reads from the same register. On many pipelined architectures, this will cause a so called pipeline stall that means that one-cycle processing is interrupted while the second instruction waits for the first instruction to finish writing to the register.

A good compiler for such a CPU architecture will try to rearrange or sched-

Listing 1

```
Bool AreIndependent(Instr inst1, Instr inst2)
{
  if (IndependentSourceAndDest(Inst1, Inst2))
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

Listing 2

```
void ChangeMOVOrder(Instr inst1, Instr inst2)
{
  // Do other processing first
 …
  if (AreIndependent(inst1, inst2))
  {
    ChangeOrderHelper(inst1, inst2);
  }
}
```

ule instructions so as to maximize the distance between instructions that use the same CPU resource in a pipeline blocking way.

To do such rearranging, the compiler must build up one or more dependency graphs for the block of instructions it's about to schedule to determine if it is safe to move an instruction backward or forward in the instruction stream. The compiler uses a set of functions to determine if two instructions are independent which means that they do not use resources in a conflicting way and thus implicates that their order can be exchanged.

Let's take a look at a function that the compiler might use to determine if two MOV instructions are independent, shown in **Listing 1**.

This function looks innocent enough. It basically shifts the question of independence to a helper function that determines if the source and destination of the MOV instructions are used independently.

It's perfectly OK for the compiler to leave the instructions together in the same order. To get the puzzle together with maximum performance as a result, it might for example sometimes just create a new pipeline stall by moving two instructions to avoid another stall.

Let's return to the function in Listing 1 and the compiler that uses this function. When a customer compiles a certain program with this compiler, it works flawlessly, except that he notes that two memory writes are done in the wrong order as opposed to how they are specified in the program. Usually this is the principle that the scheduler is dependent on to perform its magic, but in this case it's not OK because the user has specified that both variables that are affected by the MOV instructions are declared as `volatile`, which has implies that the order of the writes should not change. If, for example, the memory writes are intended to initialize some external hardware, this can be extremely important.

The `AreIndependent()` function ignores the volatile attribute of both instructions and thus reports that it's OK to rearrange these instructions.

Listing 3

```
void ChangeMOVOrder(Instr inst1, Instr inst2)
{
  // Do other processing first
 …
  // Bug detection code
  if (IsVolatile(inst1) || IsVolatile(inst2)
  {
    ReportSourceStatement(inst1);
    ReportSourceStatement(inst2);
    return;
  }
  if (AreIndependent(inst1, inst2))
  {
    ChangeOrderHelper(inst1, inst2);
  }
}
```

As noted, the scheduler can of course choose to leave two independent instructions in place. For this customer, it's easy to see that he has at least one location that is affected by this bug, but

One remedy: a special version of the original compiler can try to identify all code in the user's code base that is affected by the bug.

does he have more affected locations? Finding that out effectively amounts to going through the complete code base looking for accesses to volatile variables and examining the generated code; so we're back to the central theme of this article—how can the customer's change management be simplified?

Here is one possible remedy to the situation: a special version of the original compiler can try to identify all code in the user's code base that is actually affected by the bug.

Here is how the compiler can be turned into a bug detector. The function in Listing 1 is used in another function (**Listing 2**) that changes the order of two instructions when that function has decided that it is beneficial to do so.

This function can be changed to detect the bug case, in other words, when the `ChangeMOVOrder()` function uses the wrong information to make a decision. The added code in red in **Listing 3** looks for the offending situation and when such a situation arise it reports the affected source locations. Note how the code in red would also cure the bug because it classifies all MOV instructions with the volatile attribute as dependent. But it is crucial to understand that we could not have placed the detector code in the buggy function! If we would have done so, it would report every occurrence of possibly erroneous MOV instructions.

Even this simplified example showed us one of the pitfalls in creating a production quality bug detector. It can be simple to isolate the root cause of the bug but complicated to determine when this bug will actually result in the generation of wrong code. We could for example have a number of different functions of varying complexity that depend on the `AreIndependent()` function.

But if it's practically possible to create the bug detector, it can now be used to pinpoint the exact locations of any other code that is affected by the original bug. In this way, we can avoid going through all object code by hand to look for possible occurrences of the problem. ■

Anders Holmberg is software tools product manager at IAR Systems.

Low Cost Panel PC

The PPC-E7 is a Compact Panel PC based on a 200 Mhz ARM9 processor with the following features:

- Open Frame Design
- 10/100 BaseT Ethernet
- Real Time Clock
- I2S Audio I/O Port
- 3 USB 2.0 Host Ports
- Fanless ARM9 200MHz CPU
- 3 Serial Ports RS232/485 & SPI
- SD/MMC Flash Card Interface
- Battery Backed Real Time Clock
- Up to 64 MB Flash & 128 MB RAM
- GPIO, A/D, Timers & PWM
- Linux with Eclipse IDE or WinCE 6.0
- WVGA (800 x 480) 7" LCD with Touch
- Prices start at $495 !

Windows CE

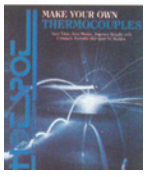
2.6 KERNEL

Since 1985 OVER 25 YEARS OF SINGLE BOARD SOLUTIONS

EMAC, inc.
EQUIPMENT MONITOR AND CONTROL

Phone: (618) 529-4525 • Fax: (618) 457-0110 • Web: www.emacinc.com

Thermocouples, Make Your Own



The Hot Spot Welder is a portable capacitive discharge wire welding unit that allows thermocouple wire to be formed into free-standing bead or butt welded junctions, or to be directly welded to metal surfaces. The HOT SPOT provides a quick, simple, accurate, low cost means of fabricating thermocouples on a "when needed, where needed" basis. Brochure and specification sheet provide photos and descriptions of thermocouple construction and use.

DCC Corp.

7300 N. Crescent Blvd., Pennsauken NJ 08110

PH: 856-662-7272 • Fax: 856-662-7862

Web: www.dccCorporation.com

FREE BOOK

on Best Practices for Peer Code Review

available at

CodeReviewBook.com

SmartBear SOFTWARE

parity bit

from page 7

a moment, you will realize that the tool is telling you: "why are you using a signed int to store a diameter for? It doesn't make sense." You will then no doubt start digging in that code piece, finding numerous other bugs.

So I daresay that any form of test would have found that bug in a few minutes, if it involved a good compiler or static analyzer. As I see it, the bug was likely caused by any combination of the following:

- A poor requirements specification. Did the specification state how thick logs that were allowed? If it did, was the product tested against that requirment?
- Poor test tools or no test tools at all.
- Insufficient knowledge in embedded programming. A qualified guess is that the program was written by an unskilled programmer who was using the default "int" type—a deadly sin in any embedded programming. But in that case the real culprit was poor coding standards at the company, or no coding standards at all.

—Lundin
R&D Manager

My experience in embedded software supports every word Mr. Grenning said.

First, in my 30 years in the field I remember only one brand-new first-release program that was a total mess, and I'm sure that was deliberate obfuscation. Most developers seem to be able to divide a program into reasonable modules and implement these modules intelligently, no matter the process they follow.

It is enhancements that destroy the programs. This was true when new breath types were added to a medical ventilator, new measurements to optical inspection systems, or new protocols to telemetry systems. These changes are almost always made under time pressure and use as much existing code as possible. The result is usually that pretty decent modules grow to be untestable and unmaintainable. I have had to take over far too much such legacy systems, and it's very hard to convince managers that it's costing them more than it's worth. If TDD can stop this code rot and force needed refactoring, I hope every organization whose code I ever have to maintain will adopt it.

P.S. Kent Beck may have taken the, write a little code, test it, and extend it, method past any reasonable point, but that is how real working programs are developed whatever the official process may be.

—The Heretic
Software Department

Editor's note: *James Grenning responds at www.embedded.com/224200702.*

We welcome your feedback. Letters to the editor may be edited. Send your comments to Richard Nass at rich.nass@ubm.com or fill out one of our feedback forms online, under the article you wish to discuss.

ADVERTISING SALES

MEDIA KIT:
www.embedded.com/mediakit

Learn today. Design tomorrow. ESD EMBEDDED SYSTEMS DESIGN

Sales Contacts

600 Harrison St., 5th Flr,
San Francisco, CA 94107

David Blaza
Publisher
(415) 947-6929
david.blaza@ubm.com

Bob Dumas
Associate Publisher
(516) 562-5742
bob.dumas@ubm.com

Advertising Coordination and Production

600 Community Drive,
Manhasset, NY 11030

Donna Ambrosino
Production Director
(516) 562-5115
dambrosi@ubm-us.com

By Jack G. Ganssle

An interview with James Grenning, Part 2

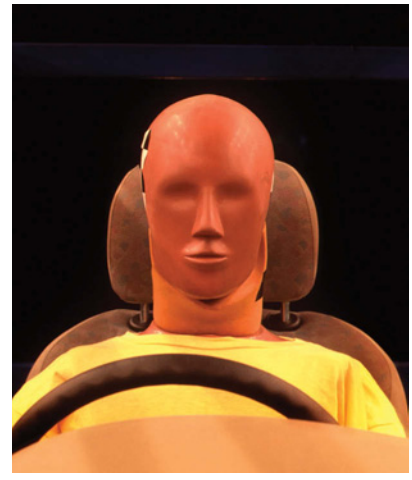
James Grenning (*www.renaissance-software.net*), whose book *Test Driven Development in C* will be out in the fall, graciously agreed to be interviewed about TDD (test driven development). The first part of our talk ran last month at *www.embedded.com/224200702*, where you can also see reader comments.

Jack: How do you know if your testing is adequate? TDD people—heck, practically everyone in this industry—don't seem to use MC/DC, npath, or cyclomatic complexity to prove they have run at least the minimum number of tests required to ensure the system has been adequately verified.

James: You are right; TDD practitioners do not generally measure these things. There is nothing said in TDD about these metrics. It certainly does not prohibit them. You know, we have not really defined TDD yet, so here goes. This is the TDD micro cycle:

- Write a small test for code behavior that does not exist
- Watch the test fail, maybe not even compile
- Write the code to make the test pass
- Refactor any messes made in the process of getting the code to pass
- Continue until you run out of test cases

Maybe you can see that TDD would do very well with these metrics. Coverage will be very high, measured by line or path coverage.



Is test driven development viable for embedded systems? It may be part of the answer. Ganssle continues to grill James Grenning on TDD.

One reason these metrics are not the focus is that there are some problems with them. It is possible to get a lot of code coverage and not know if your code operates properly. Imagine a test case that executes fully some hunk of code but never checks the direct or indirect outputs of the highly covered code. Sure it was all executed, but did it behave correctly? The metrics won't tell you.

Even though code coverage is not the goal of TDD it can be complementary. New code developed with TDD should have very high code coverage, along with meaningful checks that confirm the code is behaving correctly. Some practitioners do a periodic review of code coverage, looking for code that slipped through the TDD process. I've found this to be useful, especially when a team is learning TDD.

There has been some research on TDD's impact on cyclomatic complexity. TDD's emphasis on testability, modularity, and readability leads to shorter functions. Generally, code produced with TDD shows reduced cyclomatic complexity. If you Google for "TDD cyclomatic complexity," you can find articles supporting this conclusion.

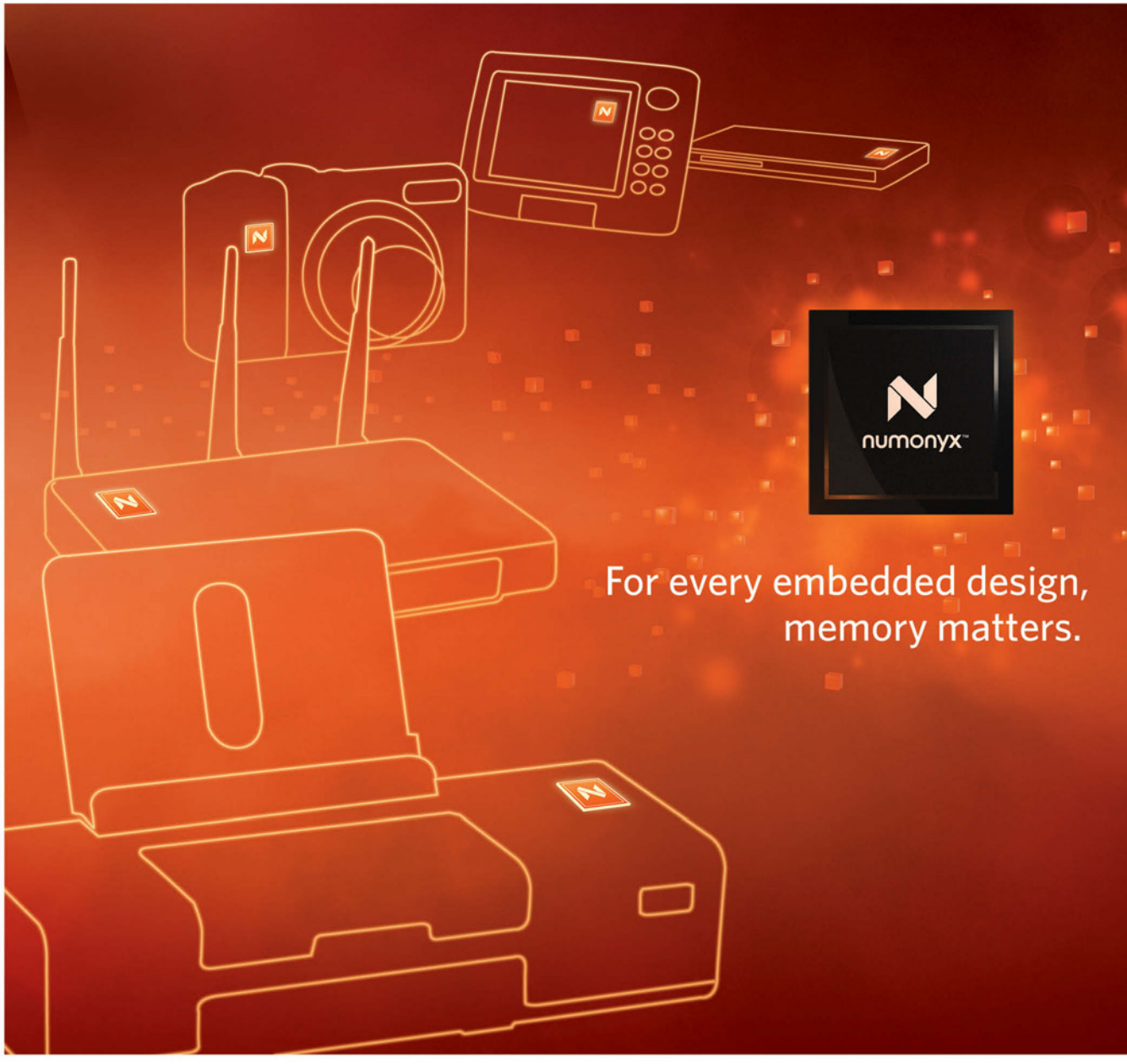
Jack: Who tests the tests?

James: In part, the production code tests the test code. Bob Martin wrote a blog a few years ago describing how TDD is like double entry accounting. Every entry is a debit and a credit. Accounts have to end up balanced or something is wrong. If there is a test failure, it could be due to a mistake in the test or the production code. Copy and paste of test cases is the biggest source of wrong test cases that I have seen. But it's not a big deal because the feedback is just seconds after the mistake, making it easy to find.

Also the second step in the TDD micro cycle helps get a test case right in the first place. In that step, we watch the new test case fail prior to implementing the new behavior. Only after seeing that the test case can detect the wrong result, do we make the code behave as specified by the test case. So, at first a wrong implementation tests the test case. After that, the production code tests the test case.

Another safeguard is to have others look at the tests. That could be through pair programming or test reviews. Actually, on some teams we've decided

Jack G. Ganssle is a lecturer and consultant on embedded development issues. He conducts seminars on embedded systems and helps companies with their embedded challenges. Contact him at jack@ganssle.com.



When it comes to creating your next embedded system, an important decision awaits you. Memory. Your design needs speed, reliability, performance and capacity to store the code and data your design demands. No problem. Numonyx has the broadest portfolio of parallel and serial NOR, NAND and phase change memory. And we offer extended temperature support with AEC-Q100 certification on many of our products and expanded design versatility with voltage support up to 5V. All designed to deliver a right-fit solution to help you shorten design cycles, reduce development costs and accelerate the roll-out of your next big idea. Find out how Numonyx memory matters for your next design.

NUMONYX® FORTÉ™ SERIAL FLASH MEMORY

Product Family	Density Range	Voltage/Solution
M25P (block erase)	512 k - 128 Mb	3V, single-I/O
M25PX (4KB block erase)	4 Mb - 64 Mb	3V, multi-I/O
M25PE/M45PE (page erase)	1 Mb - 16 Mb	3V

NUMONYX® AXCELL™ PARALLEL NOR FLASH MEMORY

Product Family	Density Range	Voltage/Solution
M29W/EW (JEDEC command set)	4 Mb - 2 Gb*	3V, page read†
P30/33 (Intel-based command set, sync burst)‡	64 Mb - 2 Gb*	1.8/3V core

Visit www.Numonyx.com/Embedded
for free access to the Numonyx Embedded Design Center.

numonyx™
innovative. memory. solutions.

© 2009 Numonyx B.V. All rights reserved.
*Includes stacked solutions †Available on specific densities ‡Other names and brands may be claimed as the property of others

that doing test reviews is more important than reviewing production code. The tests are a great place to review interface and behavior, two critical aspects of design.

Jack: As has been observed, all testing can do is prove the presence of bugs, not the absence. A lot of smart people believe we must think in terms of quality gates: multiple independent activities that each filter defects. So that includes requirements analysis, design reviews, inspections, tests, and even formal verification. Is this orthogonal to TDD approaches, and how do TDD practitioners use various quality gates?

James: TDD does not try to prove the presence of bugs; it is a defect prevention technique (*www.renaissancesoftware.net/blog/archives/16)*. People make mistakes regularly during development, but in the TDD micro cycle, the mistakes are immediately brought to the developer's attention. The mistake is not around long enough to ever make it into a bug-tracking system.

I think TDD is only part of the answer. Reviews, inspections, and pair programming are orthogonal and complementary to TDD.

There is another form of TDD, a more requirements-centric activity called Acceptance Test Driven Development (ATDD). In ATDD, the customer representative defines tests that describe the features of the system. Each iteration, the team works to complete specific stories defined by the customer. A story is like a use case, or a specific usage scenario. The acceptance tests describe the definition of done for the story. These acceptance tests are also automated. If the new and all prior tests pass, the story is done. That is an important a quality gate. Don't get me wrong, I am a proponent of reviews, but I think that TDD is superior to inspections at preventing defects.

I did a case study on the Zune bug that illustrates my point. This bug caused the 30G Zune model to freeze on New Year's Eve 2008. My informal research on the bug (*www.renaissancesoftware.net/blog/archives/38*) showed that most online code pundits who inspected the faulty function did not correctly identify the whole problem. I was in the group that got it almost right; a.k.a. wrong. Then I wrote a test. The test cannot be fooled as easy a human. So, I think we need both, inspections and tests.

Jack: Some systems are complex or control processes that respond slowly. What happens when it takes hours to run the tests?

James: For TDD to be a productive way to work, the micro cycle has to be very short in duration. This pretty much rules out going to the target during the micro cycle, and also that unit test execution must also be kept short.

If there is a lengthy control process being test driven, we need to take control of the clock. If we are managing dependencies, this is not hard.

To avoid the target bottleneck, I recommend that TDD practitioners first run their unit tests in their development system. If you are practicing the SOLID design principles it is natural to manage the dependencies on the hardware and operating system.

If there is a lengthy control process being test driven, we need to take control of the clock. If we are managing dependencies, this is not hard. A time-driven event eventually resolves to a function call. The test fixture can call the event processing code as well as some operating system, or interrupt-based event handler. If your code needs to ask some time service what the current millisecond is, we can intercept those calls and mimic any time-based scenario we like without any of the real delays slowing the test execution time.

With that said about unit tests, you might have the same issue when it comes to a more thorough integration, or system test. If you have automated some of these tests, and you rely on using the real clock, tests could take a long time to run. But that may not be a problem, because the cadence of acceptance and systems tests does not need to be as fast as unit tests. We'd like to run these longer tests automatically as part of a continuous integration system.

Jack: Let's move on to my business concerns. Through incremental delivery, TDD promises to produce a product that closely aligns with the customer's needs. That is, at each small release the customer can verify that he's happy with the feature, and presumably can ask for a change if he's not. "Customer" might refer to an end-user, your boss, the sales department, or any other stakeholder. If there's no barrier to changes, how does one manage or even estimate the cost of a project?

James: This is more of an Agile requirements management issue than TDD, but that's OK. Let me start by saying that it is a misconception that there is no barrier to requirements changes, and feature creep. For successful outcome, requirements have to be carefully managed.

In Agile projects there is usually a single person that is responsible for driving the development to a successful delivery. Some refer to this as the customer or the *product owner (PO).* The product owner might be from marketing, product management, or systems engineering. She usually heads a team of skilled people who know the product domain, the market, the technology, and testing. She is responsible for making trade-offs. Team members advise her, of course.

To manage development, we create and maintain something called the product backlog. The backlog is the list of all the features (we can think of) that should go into the product. There is a strong preference to make the work visible to the PO, over work that only engineers understand. It is mostly feature oriented, not engineering-task oriented, focusing on value delivery. We prevent surprises by taking three month engineering deliverables and splitting them

TDD is just part of the picture. The team activities should encompass cross-functional needs. While the product is evolving, the team's progress is an open book.

into a series of demonstratable bits of work that our customer cares about.

The product owner's team can add things to the backlog, but in the end, the authority of what goes into a specific iteration is the PO's responsibility. For highly technical stories, a hardware engineer might play the role of the customer. For manufacturability stories, built in test for example, a manufacturing engineer or QA person might play the role of the customer. You can see there may be many "customers," but the final call on what is worked on at what time is up to the product owner.

You also ask about estimating time and cost. There is no silver bullet here, but there is a realistic process Agile teams use. When an initial backlog is created, all the backlog items or stories are written on note cards and spread out on a table. (A story is not a specification, but rather a name of a feature or part of a feature.) Engineers get together and do an estimation session. Each story is given a relative difficulty on a linear scale. The easiest stories are given the value of one story point. All stories labeled with a one are of about the same difficulty. A story with a value of two is about twice as difficult to implement than a one. A five is about five times as difficult. I am sure you get the idea.

Once all the stories have a relative estimate, we attempt to calibrate the plan, by choosing the first few iterations and adding up their story points. We're estimating the team's *velocity* in story points per iteration. The initial estimate for the project would be the total of all story points divided by the estimated velocity. This will probably tell us that there is no way to make the delivery date. But it's just an estimate, next we'll measure.

As we complete an iteration, we calculate the actual velocity simply by adding the point values of the completed stories. The measured velocity provides feedback that is used to calibrate the plan. We get early warning of schedule problems, rather than 11th-hour surprises. If the projected date is too late for the business needs, managers can use the data to manage the project. The PO can carefully choose stories to do and not do to maximize delivered value. The business could looks at adding people before it is too late, or change the date.

Jack: Engineering is not a standalone activity. While we are designing a product, the marketing people make advertising commitments, tech writers create the user's manual, trade shows are arranged, accounting makes income and expense projections, and a whole host of other activities must come together for the product's launch. TDD says the boss must accept the fact that there's no real schedule, or at least it's unclear which features will be done at any particular time. How do you get bosses to buy into such vague outcomes?

James: Jack, there goes that misconception again on "no real schedule." There is a schedule, probably a more rigorous and fact-based schedule that most developers are used to working with. The Agile approach can be used to manage to a specific date, or to specific feature content.

TDD is just part of the picture. The team activities should encompass cross-functional needs. While the product is evolving, the team's progress is an open book. The user documentation, marketing materials, etc., can and should be kept up to date. I don't try to get bosses to buy into vague outcomes. I get bosses that are not satisfied with vaguely "working harder/smarter next time." I get bosses interested that want predictability and visibility into the work. I get bosses that want to see early and steady progress through the development cycle, ones that are not so interested in doing more of the same thing and expecting different results.

Jack: Now for a hardball question: Is it spelled agile or Agile?

James: Saving the toughest for last, setting me up. Someone with greater command of the language better take that one. Like any label, agile is aging and getting diluted. My real interest, and I think yours too, is advancing how we develop embedded software and meet business needs. To me many the ideas in Agile Development can really help teams. But its important to consider it a start, not the destination.

Jack, thanks again for the chat. It's always good talking to you.

Jack: Thanks, James, for your insightful answers. I hope the readers will respond with their thoughts and experiences using TDD in their workplace. ■

AS EMBEDDED TECHNOLOGY ADVANCES WE'RE RIGHT THERE WITH YOU

A N D R O I D ▪ N U C L E U S ▪ L I N U X

MENTOR EMBEDDED. Innovation is boundless. Mastering its possibilities requires a partner with the very latest in embedded software solutions. Mentor Embedded delivers solutions that reduce your development risk and shorten project cycles. We provide proven software and tools, along with services for Linux and Android development that enable our customers to succeed in emerging areas such as Android beyond mobile, multi-OS, multicore, and advanced 3D user interfaces. As the industry's leading independent vendor, Mentor Embedded stands ready for your next innovation. To learn more visit **www.mentor.com/embedded.**

Mentor Graphics®

©2010 Mentor Graphics Corporation. All Rights Reserved. Mentor Graphics is a registered trademark of Mentor Graphics Corporation.

R&D Prototype

PCB Assembly

$50 in 3-Days

Advanced Assembly specializes in fast assembly for R&D prototypes, NPI, and low-volume orders. We machine-place all SMT parts and carefully follow each board through the entire process to deliver accurately assembled boards in three days or less.

R&D Assembly Pricing Matrix/Free tooling and programming

Up to # SMT Parts	25	50	100	150	200	250	300	Over 300
1st board	$50	$85	$105	$155	$205	$255	$305	
2nd board	$30	$55	$65	$95	$125	$165	$185	Call for Pricing
Each additional board	$25	$35	$45	$65	$95	$125	$155	
Stencil	$50	$50	$50	$50	$50	$50	$50	

aapcb.com/esd4
1.800.838.5650

Advanced Assembly®
SMT ASSEMBLY FOR ENGINEERS™

Lead Free
SO 9001:2008 Certified

The new standard for pcb assembly